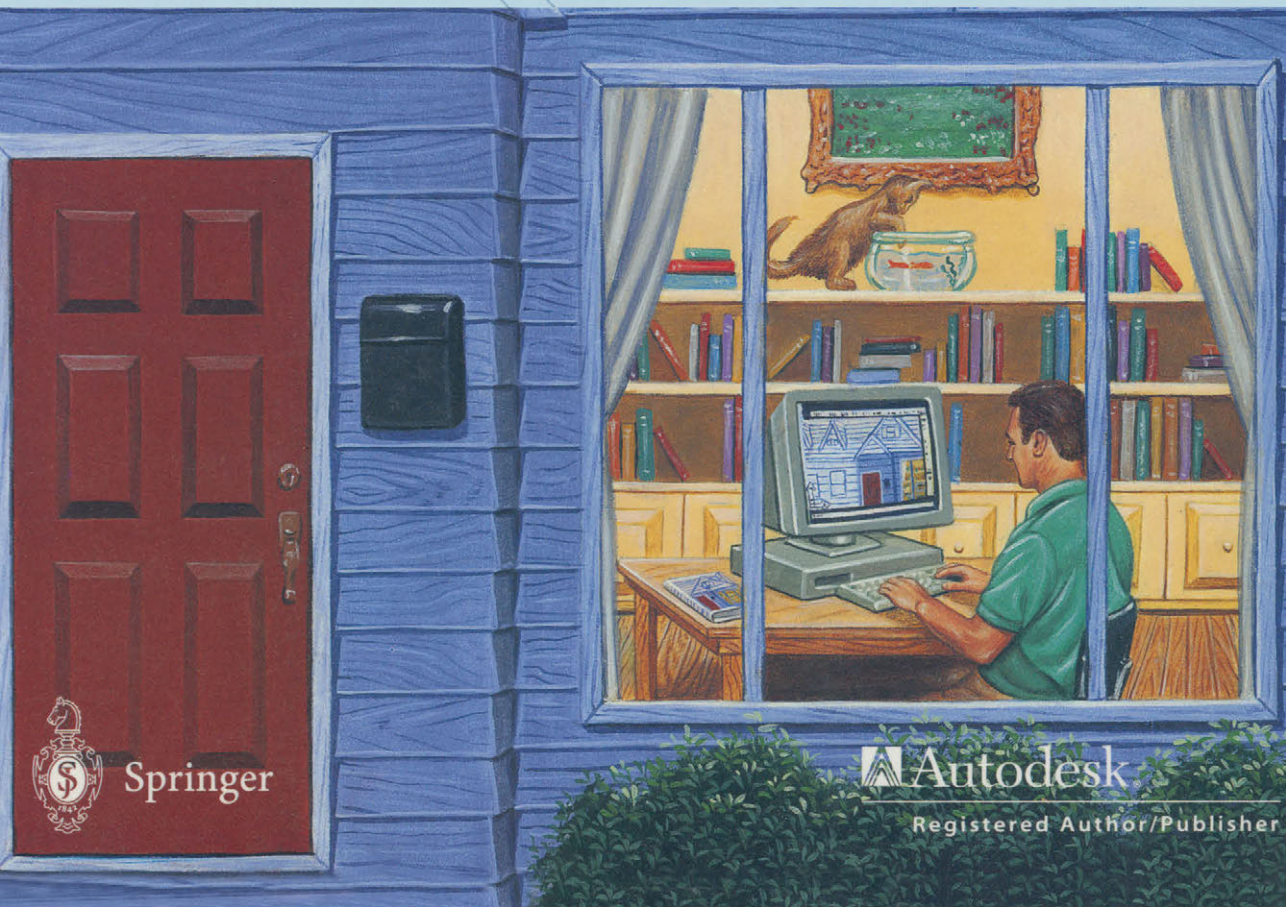
COVERS RELEASE 13 AND RELEASE 12

Roy Harkow

Essential AutoLISP®



Springer

Autodesk

Registered Author/Publisher

Essential AutoLISP®

Springer Science+Business Media, LLC

Roy Harkow

Essential AutoLISP®

With a Quick Reference Card and a Diskette

Springer

Roy Harkow
Roy Harkow Associates
40 MacArthur Road
Wellesley, MA 02181
USA
E-mail: roy@yem.pn.com

AutoLISP, AutoCAD, and AutoCAD Training Center are registered trademarks of Autodesk, Inc.
AutoCAD Development System is a trademark of Autodesk, Inc.

This book is not an Autodesk product and is not warranted by Autodesk.

Library of Congress Cataloging-in-Publication Data
Harkow, Roy.
Essential AutoLISP / Roy Harkow.
p. cm.
Includes bibliographical references and index.

1. AutoLISP (Computer program language) I. Title.
QA76.73.A84H37 1995
620'.0042'028755369—dc20 95-34192

Printed on acid-free paper.

© 1996 Springer Science+Business Media New York
Originally published by Springer-Verlag New York, Inc. in 1996
Softcover reprint of the hardcover 1st edition 1996
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

Production coordinated by Impressions and managed by Bill Imbornoni; manufacturing supervised by Jacqui Ashri.
Typeset by Impressions, a division of Edwards Brothers, Inc., Madison, WI.

9 8 7 6 5 4 3 2 1

ISBN 978-0-387-94571-2 ISBN 978-1-4612-2350-4 (eBook)
DOI 10.1007/978-1-4612-2350-4

"If my words did glow
With the gold of sunshine . . ."
—Robert Hunter

Contents

The Purpose of *Essential AutoLISP* and What Makes It Unique xiii
Why *Essential AutoLISP* Is Special if You're New to Programming xiv
Why *Essential AutoLISP* Is Special if You're New to AutoLISP xv
How to Get Started Quickly xv
Why *Essential AutoLISP* Is Special if You're Already Using AutoLISP xv
Acknowledgments xvii
Terms and Conventions Used in *Essential AutoLISP* xxi
Windows versus DOS xxiv
Release 13 versus Release 12 and Prior Releases xxiv

Part I Working with AutoLISP Data 1

1. The AutoLISP World 5
   1.1 A Short History of Lisp 6
      1.1.1 This Book's Approach to Learning AutoLISP 7
   1.2 Why AutoLISP? 8

1.3 The AutoLISP World: Understanding the Picture Notation Used in This Book 10
1.4 Numbers 12
1.5 Symbols 14
1.5.1 Values of Symbols 16
1.5.2 `nil` and `t` 18
1.6 Strings 19
1.7 Conses 21
1.8 Lists 26
1.9 Functions 34
1.10 Basic Functions 37
1.11 Important Points to Remember 52
1.12 Labs 54

2. Printed Representation of AutoLISP Objects 59
2.1 Numbers, Strings, Symbols, and Functions 60
2.2 Conses 62
2.3 Lists 64
2.4 Converting Printed Representation to Picture Notation 74
2.5 Background Topic: Rules for Simplifying Lists 83
2.6 Prefix Notation 86
2.7 The DRAW Program 91
2.8 Important Points to Remember 93
2.9 Labs 94

3. Arithmetic Functions 99
3.1 Basic Arithmetic Functions 100
3.2 Number Conversion 109
3.3 Advanced Topic: Demystifying the Bit Manipulation Functions 111
3.4 Important Points to Remember 120
3.5 Labs 121

Part II Understanding How Code Is Executed in AutoLISP 123

4. Evaluation of AutoLISP Expressions 125
4.1 How the `eval` Function Processes Our Code 127
4.1.1 How `eval` Handles Lists 128
4.1.2 Errors Reported by `eval` 132
4.1.3 Examples 134
4.2 The Eval Flowchart Program 144
4.3 The DRAW Program (Reprise) 146
4.4 Important Points to Remember 147
4.5 Labs 148

5. Writing and Running Functions 153
5.1 Special Operators 154
5.1.1 Using `defun` to Define Functions 154
5.1.1.1 The `acad.lsp` File 170
5.1.2 Using `quote` to Prevent Evaluation 171
5.1.2.1 Using `quote` with Symbols 173
5.1.2.2 Using `quote` with Lists 174
5.1.2.3 Using `quote` with Other Object Types 176
5.1.2.4 The Quote Character: ' 176
5.1.2.5 The Difference between Data and Code in AutoLISP 178
5.1.2.6 Guidelines for Using `quote` 178
5.1.3 A Closer Look at `setq` 179
5.1.4 How Code Is Executed at the Command: Prompt 180
5.1.4.1 The Read-Eval-Print Loop 182
5.1.4.2 The `1>` Prompt 185
5.2 Global and Local Variables 186
5.2.1 Creating Local Variables 187
5.2.2 Using Global Variables 191
5.2.3 Advanced Topic: Scoping of Variables 193
5.2.4 AutoCAD System Global Variables 197
5.2.5 Review Topic: Using Symbols in AutoLISP 199
5.3 Creating and Using AutoCAD Commands in AutoLISP 202
5.3.1 Running AutoCAD Commands in AutoLISP 203
5.3.2 Creating AutoCAD Commands in AutoLISP 208
5.4 Important Points to Remember 213
5.5 Labs 216

Part III Programming in AutoLISP 223

6. Control Structure: Producing Powerful Programs 225
6.1 Predicate Functions: Asking Yes/No Questions 226
6.1.1 Equality and Comparison Predicates 231
6.2 Conditional Execution of Code 240
6.3 Iteration Functions: Writing Loops to Repeat Code 250
6.4 Advanced Topic: Writing Recursive Functions 265
6.5 Important Points to Remember 271
6.6 Labs 272

7. Working with Lists 279
7.1 Basic List Functions 280
7.1.1 List Functions Quiz 280
7.1.2 Quiz Answers 282

7.2 Association Lists 293
7.3 Advanced Functions: `apply`, `mapcar`, and `lambda` 297
7.4 Important Points to Remember 303
7.5 Labs 304

8. Input/Output 309
8.1 The `get` Functions: Building an Elegant User Interface 310
8.1.1 Using `initget` to Fine-Tune the `get` Functions 321
8.2 Other I/O Functions 327
8.3 File I/O 334
8.3.1 The `load` Function Revisited 341
8.3.1.1 Automating the Edit-Load Cycle 342
8.3.2 Replacing Standard AutoCAD Commands with Our Own Commands 343
8.3.3 Obtaining Filenames via Dialog Boxes 346
8.4 Advanced File Access 350
8.5 Important Points to Remember 355
8.6 Labs 356

9. Expanding our AutoLISP Toolbox 359
9.1 String Functions 360
9.1.1 Wildcards 365
9.2 Conversion Functions 368
9.2.1 String Conversions 368
9.2.2 Number Conversions 371
9.2.3 Angle Conversions 374
9.2.4 Point Conversions 377
9.3 Measurement Functions 378
9.4 Symbol Manipulation Functions 386
9.5 Graphics Functions 397
9.6 Advanced Topic: Display Control and Device Access 399
9.7 Background Topic: Application-Handling Functions 413
9.8 Labs 415

Part IV Programming AutoCAD's Entity Database 419

10. Accessing and Modifying Entities 421
10.1 Understanding the Entity Database 422
10.1.1 Complex Entities 424
10.2 Entity Name Functions 427
10.3 Entity Data Functions 433
10.4 Handles 448

10.5 Advanced Topic: Extended Entity Data (Xdata) 450
10.6 Advanced Topic: Accessing Entities within Blocks 456
10.7 Important Points to Remember 458
10.8 Labs 459

11. Selection Sets and Nongraphical Entities 463
11.1 Basic Selection Set Functions 464
11.2 Using `ssget` to Build Custom Selection Sets 470
11.2.1 Using Filters with `ssget` 475
11.2.2 Advanced Topic: Relational Tests and Logical Operators 478
11.3 Nongraphical Entities 484
11.3.1 Symbol Tables 484
11.3.2 Advanced Topic: Dictionaries 494
11.4 Important Points to Remember 496
11.5 Labs 497

Part V Becoming Better AutoLISP Programmers 501

12. Error Handling and Debugging 503
12.1 Frequently Encountered AutoLISP Error Messages 504
12.1.1 Errors Encountered When Loading Files 505
12.1.2 Errors Reported by `eval` 507
12.1.3 Errors Discovered by Functions 510
12.2 Gotchas! 515
12.3 Error-Handling Subrs 523
12.4 Effective Debugging Techniques 526

13. Programming Style 539
13.1 Writing Readable Programs 540
13.1.1 Using Global Variables 543
13.1.2 Error-Proofing Code 548
13.1.3 Tips and Techniques 552
13.1.4 Writing Larger Programs 556
13.2 Providing Help for Commands We Write 563
13.3 Understanding Other Programmers' Code 574

Part VI Introduction to the Computer 579

14. Computer and Programming Basics 581
14.1 Computer Components 582
14.2 Data Representation and Storage 587
14.2.1 Binary Representation of Numbers 588
14.2.2 Bits, Words, and Number Representation 589
14.2.3 Boolean Algebra: Logical AND, IOR, XOR, and NOT 592
14.2.4 Storing Data and Code 595
14.2.5 Advanced Topic: Memory Management 598
14.3 Writing and Running Programs 602
14.3.1 The Edit-Compile-Debug Cycle 605
14.3.2 Calling Functions 611
14.3.3 The AHED Editor 618
14.3.4 The `acad.pgp` file 621
14.4 Labs 624

Appendixes

A. Lab Answers 627

B. ASCII Codes 691

C. Files on the Enclosed Floppy 693

D. Common Error Messages 695

E. Commonly Used Entity Group Codes 697

F. AutoLISP Function Reference 699

Index 709

The Purpose of *Essential AutoLISP* and What Makes It Unique

AutoLISP is a wonderful language. It is powerful and fun to use. Once we know how the language works we can write, debug, and modify programs extremely quickly, especially as compared with FORTRAN, C, or other high-level languages.

The way we typically learn a computer language is to read the definition of some commands, write a little program, learn some more commands, write a larger program, and so forth. Or maybe we take an existing program and modify it to fit our needs. Unfortunately this approach does not work well for AutoLISP. To use AutoLISP productively, *we must understand how the language itself works.*

One of the peculiarities of AutoLISP is that what gets printed on the screen is an *approximation* of the actual data and code. What we type in gets *altered* as it enters the computer. Thus, code that appears correct on paper can easily give rise to errors.

Part I of *Essential AutoLISP* introduces an easy-to-learn *pictorial representation* for our data and code. We'll discover how to draw pictures that *exactly* specify

the organization of data in the computer, thereby eliminating all ambiguity and confusion. Furthermore, these pictures will allow us to easily solve problems that we would otherwise have to approach by a trial-and-error method. At the conclusion of one of my workshops, students always say, "I can't imagine comprehending AutoLISP without the pictorial representation."

In Part II of *Essential AutoLISP* we learn how AutoLISP processes the code we enter. Everything we type into AutoLISP is executed by the function `eval`. To utilize the language to its fullest extent, we *must* have a strong understanding of how the `eval` function works. This book is unique in that an entire chapter is devoted to the exploration of `eval`. Why is understanding this function so important?

- `eval` often alters the code we enter, thereby executing something different from what we actually type in. This can be very confusing until we learn how the transformation occurs.

- Most error messages we receive come from `eval` or from functions called by `eval`. Many self-taught AutoLISP users report that the error messages are so cryptic that they just try different approaches until the error is solved, only to encounter the same error again a short while later.

Once we comprehend how `eval` processes our code we'll clearly understand what causes our errors. Then we can easily avoid most errors and solve many of the ones that do arise in under a minute. At the conclusion of my workshops students typically specify `eval` as the most important topic they learned during the week. *Essential AutoLISP* is the only textbook that uses the pictorial representation and presents an in-depth explanation of `eval`.

Why *Essential AutoLISP* Is Special if You're New to Programming

If you haven't programmed before, you not only need to learn the AutoLISP language, but you must learn some programming concepts as well. This book assumes no knowledge beyond basic AutoCAD and DOS or Windows, and thoroughly explains the programming steps that you need to follow in order to use AutoLISP efficiently.

Chapter 14 is a background chapter in which many basic programming concepts are explained. I don't assume that you know what a function is or how to program one. Text editing is also explained, and a text editor that works quite well with AutoLISP (and is on the enclosed floppy) is also explained. If these

concepts are new to you, peruse Chapter 14 before beginning your study of AutoLISP.

Why *Essential AutoLISP* Is Special if You're New to AutoLISP

In addition to the unique and essential fundamentals, I explain (and have included on the enclosed floppy) numerous programs that will aid your productivity. Most important, the floppy contains two **on-line tutorials** that will help you to understand and practice using AutoLISP. My feeling is that having useful programs is nice, but they never quite do *exactly* what we want, so we have to modify them anyway. It's better to understand the language really well so that we can write the programs ourselves.

How to Get Started Quickly

The best way to understand AutoLISP is to read this book cover to cover. However, since I begin by teaching fundamentals, we don't actually start to write programs until Part II. If you're impatient to get going right away, you can take the following path:

- Read Section 1.10, Basic Functions.
- Scan Sections 3.1 and 3.2 to learn how to use the standard arithmetic functions.
- Go to Chapter 5, Writing and Running Functions, and begin reading from there.

If you follow this path, please come back to Chapters 1 to 4 and learn the fundamentals as soon as you can.

Why *Essential AutoLISP* Is Special if You're Already Using AutoLISP

Through years of offering AutoLISP workshops I have found that experienced users often know a good portion of the material in Parts III and IV, in which many AutoLISP functions are explained. The focus of this book is not on learning function definitions, although to the best of my knowledge, *this is the only AutoLISP textbook that explains every function in the language.*

The information that is truly singular to this book is contained in Parts I and II, where I explain how the language works. Almost assuredly this will be new and exciting for you. Furthermore, I promise that once you understand this material you will experience an immediate and spectacular increase in your productivity.

Part V is unique as well. In Chapter 12 I not only explain the causes of most common error messages and how to solve them, but I also examine numerous other errors that don't necessarily give messages. Good programming style is the focus of Chapter 13. Why write code that just works when we can write programs that are elegant and easily understood? This is what differentiates a coder from a programmer.

You may already know the information in Chapter 14 (although it does have an interesting passage on Memory Management and presents a text editor that you might find works better with AutoLISP than your current one). On the other hand, you'll probably find the "Advanced Topic" sections sprinkled throughout the book to be of immediate use.

♦ ♦ ♦

I have programmed in numerous programming languages and believe AutoLISP to be among the easiest, and by far the most enjoyable one of all. But I found it confusing until I understood how the language really works. If you take the time to learn the fundamentals and do the exercises, I guarantee that you'll become an accomplished AutoLISP programmer and your productivity will skyrocket. I hope you have as much fun as I did exploring this exciting language.

Acknowledgments

Much of the inspiration behind the initial AutoLISP workshop came from **Mark Sturgess,** Autodesk's visionary manager who really made it all happen. At the start, **Bill Kramer** and **Tracy Weaver** got me bootstrapped up and running. Bill has graciously allowed the use of his `c:long-dist`, `c:sparse`, and `c:ss-sieve` routines. Tracy contributed the `c:new-style` command. More recently, Autodesk's Art Cooney patiently answered many questions concerning arcane aspects of Release 13.

I give thanks to **Michael Covington** and **Alan Phillips** for their permission to include the AHED and PFE editors on the enclosed floppy. I appreciate the use of these editors in my classes and for the book.

It's well known that an instructor learns from his students, and I'm grateful for the many different ideas my students have given me over the years. I enjoy seeing creative minds think of new and different ways to approach a problem, and in practically every class I present a new approach to a topic that I learned from the previous class.

A number of the programs in this book were written by students, although I had to modify most of them to fit the requirements of the book. The following programming credits are due:

David Hill	`c:ped`
Todd Moen	`c:midpoint`, `c:midln`, `c:midarc`, `c:syms`, and the idea for `c:unblk` and `c:reblk`
Tony Palmisano	`c:sclblk` and `c:blkscale`
Jim Porter	`c:zz` and `c:cc`
Ray White	`c:endit`

I'm especially indebted to Todd, who's one of the best practical AutoLISP programmers I know. I'm certain I've learned as much from him as he has from me.

I've discovered that writing a book is not a discrete task. It's a culmination of years of experience and learning. Many people who didn't directly participate in this project were instrumental in its coming to be.

Starting at Ohio State and Unisys I learned so much from **Dan** and **Nancy Saks,** and they have continued to provide both technical assistance and friendship through the years that have followed. **Kevin Pammett,** with whom I worked at DEC, is my longtime friend, adviser, and sounding board.

When I first started teaching at DEC, **Enrique Alvarez** was my mentor. From him I learned how to break down an explanation into its simplest form and to present concepts in a logical, progressive flow. This book is organized with that thought in mind: Let's learn the concepts and underlying principles first, then build on those basics.

When I joined Symbolics, my first Lisp company, I knew less about Lisp than you might know now. This book's approach to teaching AutoLISP was derived from the courses we created at Symbolics. Most of the credit is due **Allan Wechsler** and **Jonathan Balgley.** Many of the concepts and technical terms used in this book are Allan's, and as I wrote I frequently heard his voice speaking. I also thank **Bob Fischer** for inspiring me to commit deeply to a project. The idea for the Eval Flowchart on-line tutorial came from a similar program written by **Bigbob Wescott** at Symbolics.

I am pleased to give special thanks to those who helped me write this book. First and foremost, there are my book reviewers, who donated their time, efforts, and expertise to making sure this whole thing makes sense:

Keith Thorslund is a superb instructor and AutoCAD/AutoLISP consultant in Ottawa. Keith made certain that my AutoLISP explanations and examples were well presented, topical, and factually correct, and he was always available to answer my bottomless well of difficult questions. He also supplied the `c:pass` function and greatly enhanced `c:midln`.

Robert Mathews is an expert in Common Lisp and an excellent technical writer. Bob examined the technical aspects of the book down to the deepest Lisp fundamentals and corrected my grammar along the way.

Dorothy Kent is the author of the *AutoCAD Reference Guide* but an AutoLISP beginner. Dorothy shared her experience as an author, pointed out where my explanations got too technical, and made sure I emphasized the points most important to AutoCAD users.

Dr. Anthony Hotchkiss is a professor at SUNY College at Buffalo and a frequent contributor to *CADalyst* magazine. Tony's technical expertise was invaluable on some difficult topics, and his more practical approach to teaching AutoLISP provided a needed balance to my theory. He also supplied the `setv` and `rsetv` functions.

Dr. Martin Gilchrist is my editor at Springer-Verlag. I am grateful to Martin for publishing this book and for the assistance he provided during the year it took me to write it. I'd also like to thank the people at Springer and Impressions for implementing my unique design needs and turning my manuscript into a real book.

I am also indebted to two unofficial members of my review team. My good friend **Bill Skerker** knows nothing about AutoCAD or programming but a fair amount about how to express concepts clearly and enjoyably. He helped greatly in making my approach less technical and more humanistic.

Ingrid Kannel provided lots of love, support, and editing too. When I couldn't figure out some wording, Ingrid patiently reviewed the passage and got me unstuck. She's also the best children's photographer in Boston, which is why I had her take the photo at the back of this book!

Terms and Conventions Used in *Essential AutoLISP*

The following conventions are used for representing AutoLISP objects (datatypes), expressions, and keyboard entry. A familiarity with these conventions will enhance this book's readability.

- Keys you press on the keyboard: `F1`, `<space>`, `<tab>`, `<esc>`, `<enter>`
- Enter the following code as you read this example or section.

See the specified section for related information.

Here's an interesting point that may be outside the focus of this discussion.

- All code and AutoLISP objects except numbers are in code font: `defun`, `(x y z)`, `acad.lsp`.
- Symbol names always appear in the text of this book in lowercase `courier` for readability. In programming examples, symbol names are lowercase

when the user types them and uppercase when the system prints them. In the pictorial representation, symbol names are always uppercase, because that is how they are stored in the machine.

- Code font is also used for programming examples and keyboard input. When showing input at the Command: prompt, what we enter is shown in normal type and what the machine prints is in bold type. For example, we show the addition of 3 and 4 in the following manner:

```
Command: (+ 3 4)
7
```

- In the middle of an example we will occasionally provide a directive or description in parentheses, like so:

```
Command: (prompt "Select a circle:")
Select a circle: (Click on a circle)
```

This parenthetical comment, (Click on a circle), requests you to take an action rather than showing what either you or AutoLISP has printed on the screen.

- Sometimes you'll see an example with dots in the middle:

```
(defun foo ()
      .
      .
      .
```

The ellipses indicate that more code would be needed to make the function runnable, but it isn't shown. This is because it's not relevant to the point that's being explained and I'm trying to keep the example simple and straightforward.

- Function definitions are displayed in the following manner:

`write-line` str {file}	Outputs a string to the screen or to a file.

The name of the function, `write-line`, is in code font. This function accepts two arguments (i.e., data to work with), as shown by the following two words. *str* specifies that the first argument must be a string. *file* denotes that the second argument must be a file. The curly braces around *file* indicate that this second argument is optional.

`list` obj . . . obj	Returns a new list with each of the arguments as members.

In this example, `list` is the function name. *obj* means that the argument can be any AutoLISP object (i.e., datatype); there is no restriction. The ellipses (. . .) indicate that this function accepts as many arguments as we choose to provide; there is no restriction there either.

Following is a list of most datatype abbreviations used in this book:

ang	Angle expressed in radians
alist	Association list
body	One or more functions to be evaluated in order. It returns whatever the last form evaluated returns.
edata	Entity data
ename	Entity name
file	File descriptor
form	Argument given to `eval` for evaluation
func	Function
int	Integer
num	Number
obj	Any AutoLISP object (datatype)
prompt	String message sent to the user's screen
pt	Point
ss	Selection set
str	String
sym	Symbol

- The AutoLISP Reference often specifies that a function takes an optional argument that, if "non-`nil`," causes an action to occur. In this book I always use the symbol `t` to indicate a "non-`nil`" condition.

- Several places in the book we examine a program that contains functions we've not yet seen. That's unavoidable if the examples are to be at all practical. When this situation arises, I'll explain enough about the function for you to understand the example. All of the AutoLISP functions are listed in the index both alphabetically and under the heading "function definitions." User-defined functions are listed under "programs." Appendix F contains a categorical listing of all AutoLISP function definitions.

- Most chapters have labs at the end. The initial labs are tailored to present only one new function; the later ones combine functions, concepts, and techniques covered in previous chapters.

  Following a lab statement there is usually one or more examples of how to *call* the function once it has been written. These show the results you should get when you run your own function. Use the data that are presented in the examples so you know whether your solution is correct.

  Appendix A contains the answers to all of the labs. You will gain more from a lab if you attempt to solve it prior to looking at the solution. However, the answers frequently contain additional text that will aid your understanding of the concept at hand, so they're well worth reading. The lab solutions are also in the file `answers.lsp` on the enclosed floppy.

  If you don't understand what a problem is asking you to do, run the program from the `answers.lsp` file at the Command: prompt.

Windows versus DOS

Essential AutoLISP is PC-oriented. Although most AutoCAD users work in a DOS environment, more and more are moving over to Windows. For this reason I've given a slight Windows tilt to the book.

Except for appearances, there aren't many differences in AutoLISP between the Windows and DOS environments. I'll always point out where variations do occur and, if appropriate, show coding inconsistencies between both platforms.

A couple of differences between DOS and Windows should be mentioned at the start:

- In DOS we move between the text and graphics windows by pressing the `F1` key. In Windows, `F1` brings up a help window, so `F2` is used to flip screens.

- To cancel a command or function in DOS we enter `Ctrl-C`; in Windows we can press the `<esc>` key. *Essential AutoLISP* uses `<esc>` to indicate the command termination key.

Release 13 versus Release 12 and Prior Releases

This book is based on Release 13 AutoLISP, which, for the most part, is a superset of Release 12. Each new release adds functionality to AutoLISP but makes no fundamental changes to the language. With few exceptions, Release 12 programs should run in Release 13, and the programs used in this book run

on both. Most of the exceptions, of course, are programs that utilize the new Release 13 functionality.

In Release 13 the *AutoLISP Programmers Reference Manual* has been folded into the *Customization Guide.* I use the term *AutoLISP Reference* to refer to either. Similarly, the *AutoCAD Reference Manual* is now called the *AutoCAD Command Reference.* When you program, you should have these manuals nearby.

The following functions were added to AutoLISP in Release 13. Next to the topic is the section number where it is explained in this book.

`startapp`—9.7
`tblobjname`—11.3.1
`dictnext`, `dictsearch`, `namedobjdict`, and `snvalid`—11.3.2
`getcfg` and `setcfg`—13.1.1
`autoload`—13.1.4
`help` and `setfunhelp`—13.2

The entity database has been reorganized. If you are running a release prior to 13, the entity data will appear different from what's shown in this book. These differences are explained in the entities chapter (10) and summarized here.

First, the numbers used in printed representation of entity names have changed. Here's how an entity name might appear:

Release 12	*Windows*	*DOS*
`<Entity name: 60000022>`	`<Entity name: 88450510>`	`<Entity name: bb7510>`

We use the Windows version in this book.

The entity data has been changed as well; there are a few more group codes in each entity data list. In prior releases the entity data for a line might appear as follows:

```
((-1 . <Entity name: 60000022>) (0 . "LINE") (8 . "0")
 (10 2.0 3.0 0.0) (11 4.0 5.0 0.0) (210 0.0 0.0 1.0))
```

In Release 13 it looks like this:

```
((-1 . <Entity name:88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 2.0 3.0 0.0) (11 4.0 5.0 0.0) (210 0.0 0.0 1.0))
```

The extra information present in Release 13 data won't affect our programming.

I

Working with AutoLISP Data

In Part I we study how AutoLISP organizes and implements data. When we write a computer program, our first step is always to define the data to be input, manipulated, and returned by that program. Therefore, we should understand the datatypes in a programming language before attempting to use them.

Chapter 1 contains definitions and explanations of the six basic AutoLISP datatypes. What's unique about *Essential AutoLISP* is that it presents a *pictorial representation* of these datatypes. A peculiarity of AutoLISP is that the *printed representation* of AutoLISP data, that is, what the computer prints on the screen, is only an *approximation* of the actual data that are stored in the machine. Printed representation can be difficult to use because of the following:

- It doesn't always give complete information: Certain relationships among data aren't shown.

- The information can be ambiguous: Two completely independent data structures can print exactly the same way.

The pictorial representation depicts all of the data and their relationships exactly as they're stored in the computer, which makes learning and using AutoLISP much easier. After twelve years of AutoLISP programming I still draw pictures when confronted with an intricate data structure.

Conversion between the pictorial and printed representation of AutoLISP data is shown in Chapter 2. We're introduced to the DRAW program, an on-line tutorial that converts an AutoLISP data structure into its pictorial representation. Its use solidifies our understanding of the datatypes used in AutoLISP.

In Chapter 3 we learn the arithmetic functions. We see how to run expressions at the Command: prompt and obtain immediate results. This chapter serves as an introduction to Part II, in which we learn how to write our own code.

Where This Book Is Headed

Parts I and II provide an indispensable background into the workings of AutoLISP. However, we don't begin actually writing functions until Chapter 5. In the preface we suggested a path you could take if you're anxious to begin coding right away. But if you're willing to take the longer path and get to know the language really well, it helps to know where we're heading.

The purpose of AutoLISP is to increase our productivity by allowing us to write programs that automate tasks that would be quite time consuming in AutoCAD alone. To see an example, let's run a program that we'll write later in this book.

Insert the diskette that comes with this book into the disk drive and load the file `answers.lsp` at AutoCAD's Command: prompt. If the floppy is in the `a:` drive, enter

```
Command: (load "a:answers")
```

We'll run a program that allows us to scale blocks. It's flexible enough to scale randomly selected inserts or all occurrences of a selected block. Before running the program, create a couple of different blocks and insert them in the drawing several times. Then run the `blkscale` command as follows:

```
Command: blkscale
Change all occurrences of the Block? (Yes/<No>): y
Select one of the Blocks: (select a block)
Scale Factor (1 equals no change): 1.5
```

At the conclusion of this command, all inserts of the selected block will be scaled 50 percent larger. Now run the command again and this time answer "No" to the first question. You'll then be able to select whichever individual inserts you want scaled.

This is an example of the time-saving commands you'll be able to write by the end of this book. Don't lose sight of this fact as you learn the essential background information that will make you a truly productive AutoLISP programmer.

1

The AutoLISP World

Topics Covered in This Chapter

- An in-depth exploration into how AutoLISP implements *data*
- The properties of six commonly used datatypes and their relationships with one another
- How these datatypes are constructed and used
- Certain specific object relationships, as shown in the following chart:

AutoLISP Objects	*Object Relationships*
Numbers	
Symbols	values
Strings	
Conses	cars, cdrs
Lists	members, elements
Functions/Subrs	

Goals for This Chapter

- Recognize each AutoLISP object by its picture notation.
- Understand the properties of each datatype, its purpose, and how it is used.
- Learn to move through and manipulate data structures via the values of symbols and the cars and cdrs of conses and lists.
- Understand the notion of a function's "contract." Learn the contracts of a few basic functions.

1.1 A Short History of Lisp

Lisp was developed in the late 1950s by John McCarthy at MIT and, excepting FORTRAN, is the oldest high-level computer language that is still widely used. He chose the name *LISP* as an acronym for ***LIS***t ***P***rocessing, because much of what we do in the language involves operations on lists of data and code.

Whereas FORTRAN was immediately embraced by the scientific community and established as the standard programming language, Lisp lay dormant for many years. One reason was that Lisp didn't address numerical computation very well, which is FORTRAN's primary strength. Lisp runs comparatively slowly and requires vast amounts of memory, and processor speed and computer memory were resources in short supply during the 1960s and 1970s (in fact, McCarthy wasn't able to demonstrate the language until computer memory quadrupled from 8 to 32 *kilobytes!*). However, small pockets of Lisp interest developed and flourished in research and technology communities.

During the years of Lisp dormancy, various practitioners developed different dialects of Lisp to fit their needs. East Coasters used MIT's version, MacLisp, while many on the West Coast ran a version called InterLisp, which took root at Xerox Palo Alto Research Center. Franz Lisp was created at Berkeley and grew popular in Europe. NIL, X, and other versions were used around the world.

By the late 1970s memory had become cheap enough and processor speeds fast enough to make Lisp useful for practical applications. The Artificial Intelligence (AI) Laboratory at MIT developed a very powerful workstation dedicated to running the ZetaLisp dialect. In 1981 AI Lab members branched out and formed two companies, Symbolics Inc. and Lisp Machines Inc., to market their machine. Lisp finally became a commercially viable language, and its popularity soared.

In the 1980s the focus of Lisp was in two areas: AI and expert systems. As its name suggests, *artificial intelligence* is the field that endeavors to get a computer to "reason," draw conclusions based on assumptions, and recognize visual patterns (for, say, robotics).

Expert systems is the practical side of AI; it attempts to embody the capabilities of an expert in some field in the computer. For example, it would be quite helpful if a rural doctor without easy access to a needed specialist could enter a patient's background and symptoms into the computer and receive a diagnosis and some suggestions for a remedy.

During Lisp's ascendancy, many companies began creating powerful new products in a variety of dialects. Realizing that a standard language would have to be agreed upon, the major players met and developed a new language that incorporated the best features of the various dialects. Thus was born Common Lisp. This is the Lisp version that is most widely used by the AI and expert systems community today.

In the mid-1980s Autodesk's developers decided to create a language to run under AutoCAD and chose Lisp for the reasons discussed in Section 1.2. They found that an early Lisp dialect called XLISP, which was developed by David Betz to run on PCs, best fit their needs. They removed from this version many functions that were not essential to the running of AutoCAD, in order to make the language more compact. They then added functions, such as the `gets`, that interact closely with AutoCAD. AutoLISP first appeared in AutoCAD Version 2.18, released in January 1986. It has been enhanced in every release of AutoCAD since then, resulting in the powerful AutoLISP language that we're using today.

Conceptually, AutoLISP is akin to the other Lisp dialects. Once we know AutoLISP we can easily learn Common Lisp. Many of the functions in the language are dissimilar, however, and the differences are quite interesting. When appropriate, we will contrast AutoLISP and Common Lisp, and several of the exercises have us write useful functions that are present in Common Lisp but absent from AutoLISP.

1.1.1 This Book's Approach to Learning AutoLISP

The approach taken in this book (as in my AutoLISP workshop) is radically different from the approach taken by any other AutoLISP book or course that I am aware of. It is described fully in the preface to this book. Please read that section so you understand the *raison d'être* of my unique methodology and gain the motivation to learn these important concepts.

I did not invent the teaching methods used in this book. They were first developed at MIT by Bernard Greenberg, Dan Weinreb, and Alan Bawden. They were fleshed out and perfected at Symbolics, primarily by Allan Wechsler and Jonathan Balgley. What's important is that this approach has been proven over time to be the best way to learn Lisp and to be able to use it productively.

When I explain functions, I frequently provide one or more examples. You'll find the explanations more revealing if you type in these examples at the Command: prompt. As you do, keep the following in mind: AutoLISP is reloaded every time we open a new drawing, and nothing we do in AutoLISP can

irreparably harm our drawing, AutoCAD, or AutoLISP itself. Therefore, never hesitate to try something new.

1.2 Why AutoLISP?

There are many different computer languages. Why did Autodesk decide to implement a version of Lisp rather than C, BASIC, FORTRAN, or one of the others?

1. First, and foremost, AutoLISP is *interactive.*

   When we enter an AutoLISP function at the Command: prompt (or load it from a file), it is executed immediately. This has a profound effect on how we program.

   Most languages, such as FORTRAN and C, require their programs to be compiled, which is a time-consuming process. Therefore, when using these languages we tend to write long programs, then compile and debug them in their entirety. (Compilation is explained in Section 14.3.1.)

   Since AutoLISP is interactive, we can create a short program (say, just two to six lines) and run it right away. Once the errors have been removed, we can write a few more lines of code and run the program again. This incremental approach to programming is decidedly faster because we can add little bits to our program as we go along rather than having to put an entire program together all at once. Furthermore, it's generally much easier to debug a six-line program than it is to debug a six-page program.

   AutoLISP's interactivity provides another advantage. Suppose we have a program that someone else has written and we want to learn how it works so we can modify it. With most languages our only recourse is to read the code. In AutoLISP we can type the program, line by line, at the Command: prompt and see exactly how the program behaves. This technique is quite useful and is shown in detail in Section 10.3.

2. AutoLISP allows us to represent *various types of data* in a single list.

   Most computer languages allow us to create, say, an array of real numbers, or a structure consisting of of strings, but they cannot mix real numbers and strings in a structure that continually changes size. This is one of AutoLISP's main features. Why is that important to us?

AutoCAD keeps track of every entity that we draw. Each circle, arc, and text entity in our drawing is represented in an AutoCAD database. Let's examine some of the information that might be stored for circle:

`"CIRCLE"`	A specification of the entity type
`"0"`	The layer that the circle resides on
`(2.0 3.0 0.0)`	The center point of the circle
0.4	The circle's radius
1	The circle's color code if it is not BYLAYER

Entity data can be composed of real numbers (radii), integers (color code), strings (layer name), lists (center point), and several other datatypes. Most languages don't allow mixing datatypes in one structure, but Lisp is very flexible in this regard.

3. AutoLISP enables us to express *relationships* among the objects that we create.

   Every language has a *forte*. FORTRAN is a ***FOR***mula ***TRAN***slating language; it performs mathematical operations very quickly. COBOL is the ***CO***mmon ***B***usiness ***O***riented ***L***anguage; it manipulates large databases with ease. LISP is an acronym for ***LIS***t ***P***rocessing; it's best at building relationships among items in lists of data.

   For example, AutoLISP allows us to write code for a window and have that code apply to the windowsill, panes, and curtains as well. Other languages would treat these as unrelated pieces of data. As with AutoCAD's Block or Group capabilities, it is often advantageous to be able to relate distinct objects together so that we can perform tasks on the entire group.

In this chapter we are going to examine the object types (i.e., datatypes) that exist in the AutoLISP world and learn how to create relationships among them.

This is a fundamental study of how AutoLISP implements data. In Part II of this book we'll examine how AutoLISP implements actions. Only after we have an understanding of the structure of AutoLISP can we write clean, efficient, error-free code.

1.3 The AutoLISP World: Understanding the Picture Notation Used in This Book

The AutoLISP language has several different datatypes. Some of these, such as integers and real numbers, are familiar to us all. Others, such as symbols and strings, are well known to programmers. Still others are new even to those who have programmed in other computer languages.

In AutoLISP, data and code look very much alike. The term *datatype,* used in the *AutoLISP Reference,* can be misleading, because it leads us to think in terms of data only and forget about the code. For this reason I introduce the term *object type* as a synonym for *datatype.* Furthermore, I'll use the term *object* to refer to a particular instance (say, an individual number or string) of a given object type.

When we input data or code into the computer, we use a *printed representation* of that data or code. AutoLISP uses the same printed representation to output information to our screen. Unfortunately, this printed representation is only an *approximation* of how the data and code are actually stored in the machine; it does not depict data and code completely or even totally accurately. We'll see the problems of printed representation as we examine how the data are organized in the language.

It is sometimes very difficult for a new user to look at an AutoLISP routine and understand what that code is doing. Expressions are frequently nested many levels deep, and their clarity is influenced by the spacing between each. Furthermore, AutoLISP often *alters* what we type in, so what we see on the screen is not necessarily what is actually stored in the machine!

In this book I use a *pictorial representation* of the various AutoLISP object types. It provides the following advantages over printed representation (which may not be meaningful to you just yet but will be in a very short time):

- Pictorial representation more clearly depicts the relationships among AutoLISP objects and more accurately portrays the organization of data in the machine than the printed representation can.

- Pictorial representation shows the values of symbols; printed representation doesn't.

- Two different lists may have the same printed representation; the pictorial representation will differentiate between them.

- Given a typically intricate list, such as `((a (b c) d) e)`, it is very hard to ascertain what functions are needed to access the symbol `d`. The pictorial representation makes this determination trivial.

As we shall repeatedly see, the pictures not only facilitate our learning of AutoLISP but help even experienced AutoLISP programmers to visualize intricate data structures and code. Therefore, the paradigm taught in this book will be of great assistance to you throughout your AutoLISP programming career.

In this chapter the various AutoLISP object types are introduced using our pictorial representation, which enables us to better grasp the concepts behind each individual object. In the next chapter the printed notation is explained, and we see how to convert from the picture notation to the printed representation and back again.

The collection of pictures that depict the AutoLISP code and data is called the *AutoLISP world.* Figure 1-1 shows some of the object types that "live" in the AutoLISP world:

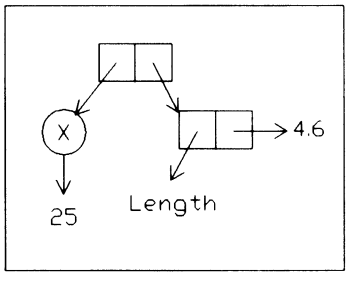


Figure 1-1

The remainder of this chapter is devoted to the examination of the object types that exist in the AutoLISP world. Since both the data and the code are composed of these objects, it is important that you understand the properties of each of them, and how they relate to each other.

Not all object types are discussed in this chapter. The more advanced object types are explained in the following chapters:

Files	Chapter 8
Entity names	Chapter 10
Selection sets	Chapter 11

1.4 Numbers

Numbers are used in AutoLISP in much the same manner as they are used in AutoCAD. We count with them, perform mathematical operations on them, use them to represent colors, and convert them to strings in order to place them in our drawings. Those of us who have programmed in other computer languages will find that numbers behave in a familiar manner.

There are two types of numbers in AutoLISP: integers and reals. Figure 1-2 shows the various ways that numbers can be depicted:

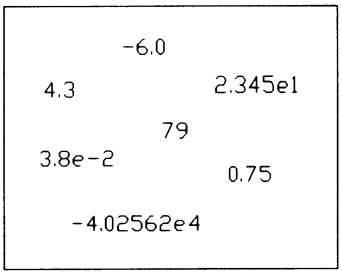


Figure 1-2

Integers

Integers are the counting numbers, such as 0, 1, 2, 3, –1, and –25. Within AutoLISP, the largest integer is 2^{31-1}. However, when numbers are passed between AutoLISP and AutoCAD, the largest integer is 2^{15-1}. Thus our practical range for integers is –32768 to 32767, the maximum that can be expressed in a 16-bit word. (An explanation of number representation is presented in Section 14.2.2.)

Numbers larger than 32767 must be expressed as real numbers, which are stored in 32 bits.

It is best to use integers wherever possible because they take less memory than real numbers and can be accessed in less time. However, when we are doing precise calculations, performing divisions, or working with points, real numbers are necessary.

Real or Floating Point Numbers

Real numbers are numbers that contain a decimal point. They have an integer part to the left of the decimal point and a fractional part to the right. *Floating point* is another name for real numbers that is used in programming languages.

Real numbers can be expressed in two different ways. The first is using the decimal notation that is familiar to us all. Examples of a real number expressed in this fashion are 4.3, –8.223, –6.0, and 0.75.

When we want to express ¾ as a real number, we must enter 0.75; the leading 0 is required. If it's omitted, AutoLISP returns an "invalid dotted pair" error.

The other way to express a real number is by using *scientific notation.* This notation appears on our screens from time to time and is used if we select "Scientific" in the AutoCAD UNITS command, so it's beneficial to be able to recognize it. But if you don't use scientific notation in your work, it is not crucial that you understand the following explanation.

Numbers are usually expressed in scientific notation with one digit to the left of the decimal point and the necessary number of digits to the right. This number is followed by the letter *e* and an integer. The integer expresses the power of 10 to which this number is to be raised.

For example, the number 23.45 would be expressed as 2.345e1; when 2.345 is multiplied by 10 the result is 23.45. Similarly, the representation of –40256.2 would be –4.02562e4. In this case the number –4.02562 is multiplied by 10^4, or 10,000.

We can use negative *e* values to indicate a division by 10: 3.8e–2 is equivalent to 0.038, which is 3.8 divided by 10^2, or 100. The term *floating point* arose because the decimal point moves or "floats" left or right depending on the *e* value.

Real numbers are stored internally in a *double-precision* format, which means that two words (a full 32 bits) are used to hold a number. This obviously allows numbers to be much more precise than if only one word was used to store their magnitude.

Unfortunately, floating point arithmetic on a computer is notoriously imprecise. From time to time when we expect the result of a calculation or even a coordinate of a point to be 0, it may come back as –3.20885e–17. This is as close as the computer could come to 0 for this calculation, and the number is so small that for all practical purposes it is 0. Don't be surprised or concerned when you see such a number . . . it won't noticeably affect our work.

We can convert numbers into strings in a format similar to the ones provided by the AutoCAD UNITS command, then place them either in our drawings or in a message to be printed on the screen. (Number conversions are covered in Section 9.2.2.)

One number is special in AutoLISP. This is the number that is the value of the variable `pi`. `pi` is defined to be approximately 3.14159; the number of digits we see to the right of the decimal point is controlled by the UNITS command. `pi` is actually defined much more precisely than what we see printed on our screens, and even more precisely than we are able to redefine it ourselves. *Never redefine the value of* `pi` !

1.5 Symbols

Symbols serve two important purposes in AutoLISP:

1. They are used as *variables* to represent known or unknown quantities.

2. They are used to name AutoLISP objects that wouldn't otherwise have names. Some objects don't have inherent ways of being accessed, so we "name" them with symbols.

Symbols have two attributes that we need to study:

1. Every symbol has a name. This is how we identify and access the symbol.

2. Every symbol has a value. This is how we use the symbol.

We'll examine symbol names first, then in the next section we'll see how symbols are used.

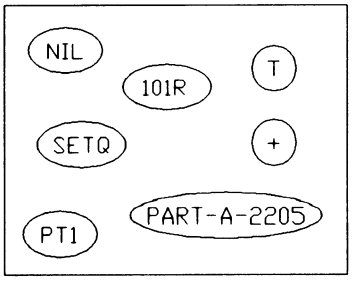


Figure 1-3

In our pictorial representation, we always depict a symbol by writing the name of the symbol within a circle or an oval. Anything inside a circle is, perforce, a symbol name. Symbol names can be any combination of letters, numbers, and special characters, with a few exceptions:

- A symbol's name cannot be all digits. In other words, `101r` is a valid symbol name, as is `r101`. But 101 is a number and can never be used as a symbol.

- The following special characters are illegal in symbol names because they have other uses in AutoLISP:

  ```
  ( ) . ' " ;
  ```

  It is illegal to put spaces in a symbol name. For example, we cannot have a symbol `a b`; AutoLISP assumes that these are two separate symbols. When we want a symbol to have more than one word in its name, we separate each word with a hyphen. Thus `part-a-2205` is a valid symbol name.

We try to give symbols names that somehow indicate their purpose. For example, `sum` is a meaningful symbol name, but `qqq` (or your own name, for that matter) usually is not.

Symbol names can be any length, which is refreshing, because most computer languages impose a limit of six or eight characters. In its Memory Management chapter, the *AutoLISP Reference* states that we should limit our symbol names to six characters or less in order to save space. This isn't necessarily true, for the following reason:

We almost always give symbols short names so that they require less typing. For example, we'd typically use symbols such as `pt1` and `pt2` to represent points. On occasion we have the need for a longer name in order to truly describe the purpose of a symbol. A program containing even a dozen long symbol names would use only a few extra words of memory. Thus, unless there is a need to conserve every possible memory word, we're better off using a longer symbol name when it increases the clarity of our code.

Symbol names are not case sensitive. It does not matter whether they are entered in uppercase, lowercase, or mixed case. Internally, all symbol names are converted to uppercase. When the machine prints a symbol name on the screen, it *always* appears in uppercase. Conversely, any uppercase word printed by AutoLISP must be a symbol (unless it is encased in double quotes, in which case it's a string).

Symbol names, such as `pt1`, always appear in the text of this book in lowercase, bold type for readability. In the pictorial representation (shown in Figure 1-3), symbol names are always uppercase because that is how they are stored in the machine.

1.5.1 Values of Symbols

Every symbol has exactly one value. A *value* is a relationship between a symbol and another AutoLISP object. In our pictorial representation we represent a symbol's value via an arrow going *from* the symbol *to* its value.

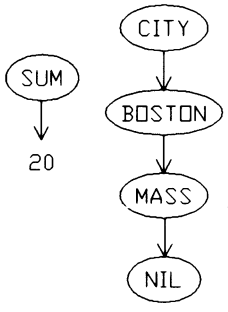


Figure 1-4

Any AutoLISP object can be the value of a symbol. In other words, a symbol's value can be a number, another symbol, or any one of the other AutoLISP object types that we haven't yet seen. When we use a symbol to represent an AutoLISP object that doesn't have a name, then that object becomes the value of the symbol. When we use a symbol as a variable, the value of the symbol is the quantity that the symbol represents.

In Figure 1-4, the value of the symbol `sum` is the number 20, the value of the symbol `city` is the symbol `boston`, the value of `boston` is the symbol `mass`, and the value of `mass` is the symbol `nil`. Because the value of `sum` is 20, we often say that `sum` is *bound* to 20.

There is a special symbol in AutoLISP, named `nil`, that has a variety of uses. This symbol will be discussed in detail shortly, but one aspect of `nil` is relevant here: If we create a symbol and do not explicitly assign it a value, then the value of that symbol automatically becomes `nil` (it is impossible for a symbol to have *no* value). In the preceding diagram, the symbol `mass` has `nil` as its value. In this case we say that `mass` is *unbound.*

A symbol is created when we first use it; we don't have to "declare" it, as in some other languages. Furthermore, many symbols can be bound to `nil` in the AutoLISP world. Therefore, we use the following convention to simplify our pictorial representation:

> Whenever we draw a symbol whose value is `nil`, we don't draw the `nil`.

In other words, when we see a picture of a symbol that appears to have no value, we should immediately realize that the symbol's value is `nil`. When a symbol is created, its value is `nil` until another value is assigned to it. In Figure 1-5, the symbols `a` and `b` have been assigned values, but `c` is still unbound:

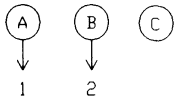


Figure 1-5

Symbols can be utilized algebraically to represent quantities known or unknown. When used in this manner, they are called *variables.* In algebra (or other computer languages such as FORTRAN or BASIC), variables are used in the following way:

```
x = 10       Set the value of x to 10
sum = 0      Set the value of sum to 0
sum = sum + x  Add the value of x to the value of sum
```

These three expressions can be written in AutoLISP as follows:

```
(setq x 10)
(setq sum 0)
(setq sum (+ sum x))
```

The way we write these expressions may seem odd at first but will quickly become familiar. `setq` is the AutoLISP function that assigns initial values to variables or changes variables' values. It has the same purpose as = in some other languages (= is an AutoLISP function as well, but its usage is different).

Throughout this chapter we present examples using `setq` to provide a frame of reference to those of you who have used AutoLISP. If you're new to AutoLISP, it's not important that you understand these examples the first time you encounter them. setq is defined in Section 1.10.

As we have seen, we use an arrow to depict a symbol's value by drawing the arrow from the symbol to its value. Two other datatypes, *conses* and *lists,* also use arrows. We have two important rules that apply to arrows *wherever they appear,* be it in symbols, conses, or lists:

1. Don't backtrace arrows.

   When determining relationships among AutoLISP objects, we always proceed in the direction that the arrow points; we never move in the opposite direction.

Assume that the relationships shown in Figure 1-4 have been created in the AutoLISP world. When we ask for the value of the symbol `city`, `boston` is returned. If we then ask `boston`, "What's your value," it replies `mass`. But if we ask `boston`, "What symbol are you the value of?" it can't tell us.

A symbol (or any other AutoLISP object) does not know what symbol it is the value of. Remember that `nil` is the value of *every* symbol that has not had a value explicitly assigned to it. If we asked `nil` what symbol it is the value of, it would be unable to tell us because it's the value of many symbols.

Although this may seem peculiar at first, we do not lose programming power by not being able to backtrace arrows. In practice we often have to find the value of a symbol but almost never need to inquire about the reciprocal relationship. On the rare occasions when we do, there are ways to easily work around this apparent limitation. For example, it is entirely permissible to have a symbol, `x`, whose value is the symbol `y`, and make `y`'s value be `x`. However, this is generally not a very useful relationship to create. More often than not, if we need a symbol once we've derived its value, we simply remember that symbol's name.

2. Don't trace through arrows.

   In Figure 1-4 the value of `city` is `boston` and the value of `boston` is `mass`. The symbol `city` knows *nothing* about the symbol `mass`.

   When several objects are connected by arrows, *never* trace the arrows through to the end. Recall that the value of `mass` is `nil`. If we trace the arrows from `city` all the way to the end, then we actually end up with `nil`. By that logic the values of `city`, `boston`, and `mass` would all be `nil`, and there would be no purpose in creating the relationships in the first place!

   If we have the symbol `city` and we want to get the symbol `mass`, we must first find the value of `city` and then find the value of that result. To say it another way, we must find the *value of the value* of `city`; it takes two independent steps to move along two arrows. Again, keep in mind that this is true not only for symbols, but also for any object types that use arrows to build relationships.

1.5.2 nil and t

AutoLISP has a couple of special symbols, called `nil` and `t`, that behave in a different manner from other symbols. We have already encountered `nil`, but

there is more to it than we have seen thus far. These are the traits peculiar to `nil` and `t`:

- The value of `nil` is always `nil`. AutoLISP does not allow us to change its value.
- If we create a symbol and don't explicitly give it a value, its value defaults to `nil`. The symbol is then said to be *unbound.*
- `nil` does *not* mean *nothing* or *zero;* it is simply a defined symbol in AutoLISP.
- `nil` is used to express the *false* condition in true-false tests or the *no* condition in yes-no tests. `t` is often used to express the *true* or *yes* condition.
- Like `nil`, `t` is a predefined symbol in the language. The value of `t` is initially `t`, but its value can be changed. Except for the fact that its value has been initialized to `t` (rather than `nil`), `t` behaves like a normal symbol.

  We have a tendency to use the symbol `t` in our programs to stand for such quantities as *tolerance* or *temperature.* If we alter the value of `t` we can get into trouble, especially if we happen to make its value be `nil`. The odds of encountering a problem are slim, but it's still best to avoid changing the value of `t`.
- One definition of `t` is non-`nil`. When the *AutoLISP Reference* specifies non-`nil` data, anything that's not `nil` will do, but `t` is preferred. *Essential AutoLISP* always uses `t` in these situations.

1.6 Strings

A *string* is a group of characters encased in double quotes. Sample strings are shown in Figure 1-6:

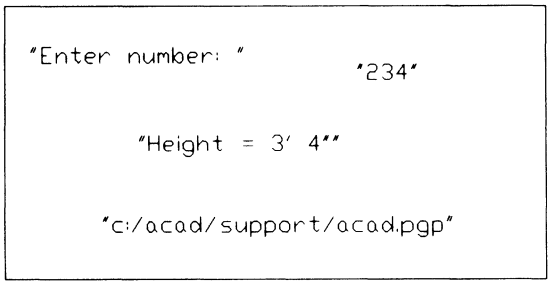


Figure 1-6

Any character can be a member of a string, including spaces; strings don't have the same name limitations as symbols. Furthermore, a character's case is preserved in a string. If we enter a string in lowercase, it stays lowercase; if we enter it in mixed case, it stays that way as well. Strings are limited to 132 characters, however. An attempt to create a longer string will result in an "exceeded maximum string length" error.

Digits can be members of a string, but they have no meaning as numbers. The string `"234"` must be converted to an integer before arithmetic operations can be performed on it.

When used in an AutoCAD command, the string `"234"` is automatically converted to the number 234 by AutoCAD.

Strings have several purposes in AutoLISP:

- When we want to put information in our drawing, or send a message to the terminal screen, strings are the preferred means of doing so. The other way that information can be printed is with symbols. Since symbols always print uppercase and cannot contain embedded spaces, strings are more attractive.

- We use strings to specify filenames. When we need to supply AutoLISP with a filename or a path to a file, we must encase that name in double quotes. In Figure 1-6, `"c:/acad/support/acad.pgp"` is a pathname to a file.

  Pathname components are separated by a backslash (\) in DOS. But in AutoLISP, backslash has an alternate use. Therefore, we separate DOS pathname components in AutoLISP by either a double backslash (\\) or a forward slash (/).

- A third place where we use strings is in AutoCAD commands. When we call an AutoCAD command from within AutoLISP, we often supply the data that the command needs as strings.

We can make a string be a value of a symbol with `setq`:

```
Command: (setq part "Part N.32.A")
"Part N.32.A"
```

However, strings cannot themselves be used to build relationships. In other words, a string can have many arrows coming in, but it can't have any arrows going out. Since strings can't be used to build relationships, they aren't very interesting from a programmer's standpoint.

1.7 Conses

A *cons* (plural: *conses*) is the object type that most differentiates Lisp from other computer languages. AutoLISP's *forte* is building relationships among objects, and the cons is the tool we use to build them. Its name is short for *construct,* because a cons is the basic building block for constructing large code and data structures.

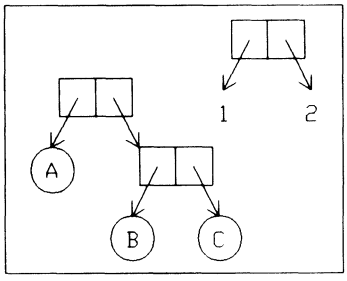


Figure 1-7

In our picture notation we represent a cons as a rectangular structure with a vertical line down the middle that separates the left half of the cons from its right half. An arrow coming from each side points to another AutoLISP object.

The AutoLISP object that the left-hand arrow points to is called the *car* of the cons. The AutoLISP object that the right-hand arrow points to is known as the *cdr* (pronounced *"could-er"*) of the cons.

Every cons *must* have both a car and a cdr. It is impossible to create a cons without both a car and a cdr, and we cannot change or remove either once the cons has been created. Let's examine a few conses and see what their cars and cdrs are:

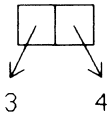


Figure 1-8

The car of this cons is 3; its cdr is 4. This is where the left and right arrows point.

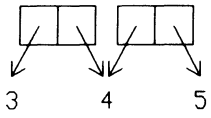


Figure 1-9

The car of the cons on the right is 4; its cdr is 5. 4 can be both the cdr of one cons and the car of another. It can be the value of one or more symbols as well:

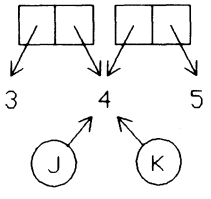


Figure 1-10

Every cons must have exactly two arrows going out from it. Every symbol must have exactly one arrow emanating from it (although we don't show the arrow if the symbol's value is `nil`). But any AutoLISP object can have an unlimited number of arrows going *into* it. Furthermore, it matters not a whit in which direction the arrows point. Figure 1-11 shows a valid cons:

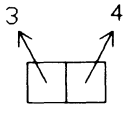


Figure 1-11

We cannot access a portion of a cons. In Figure 1-12 the symbol a points to the left half of the cons, but its value is the entire cons.

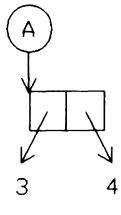


Figure 1-12

The left box in this figure is *not* the car of the cons. This cons' car is 3.

Now let's examine some conses whose cars aren't numbers.

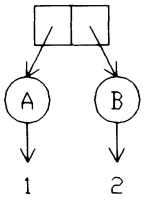


Figure 1-13

The car of this cons is `a`, its cdr is `b`. The fact that `a` and `b` are bound is irrelevant; we don't trace through arrows to the 1 and 2.

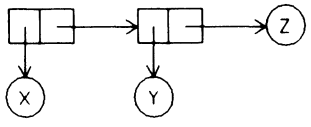


Figure 1-14

The car of the cons on the left is `x`. What's its cdr? If you said `y`, you traced through arrows. The right-hand arrow points to the cons' cdr, which is the cons on its right. That cons, in turn, has a car and cdr of `y` and `z`, respectively. Examine Figure 1-15:

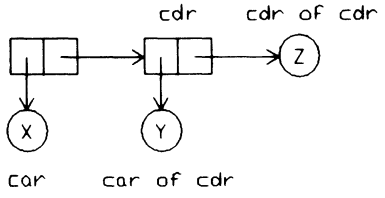


Figure 1-15

It takes two arrows to go from the cons on the left to `y`. Therefore, it takes two steps to get there. Since `y` is the car of the cons on the right, and that cons is the cdr of the cons on the left, `y` is the *car of the cdr* of the cons on the left.

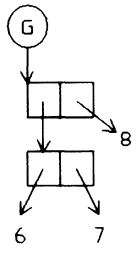


Figure 1-16

What are the car and cdr of the cons on top? Its car is another cons and its cdr is 8. If you said that the car is 6, then you traced through arrows. 6 is actually the *car of the car* of the top cons. Again, it takes two steps to go two arrows.

In the same picture, what's the *cdr* of the *car* of the *value* of g ? To answer this question, we must first determine the *value* of the symbol g. In this picture, g's value is the top cons.

Next we must learn the *car* of that cons. The car is the cons below.

Finally, we need to determine the *cdr* of the lower cons; it's the number 7. Thus, the cdr of the car of the value of g is 7. This brings us to one of the most unusual rules of AutoLISP.

In AutoLISP, we do everything backward!

This concept appears strange when we encounter it for the first time, but most people get used to it fairly quickly. What's important is that *everything* works this way, so that once we get accustomed to working backward, it becomes second nature.

AutoLISP is a very *regular* language. It has many rules with very few exceptions. Once we learn these rules AutoLISP is a very easy language to program in.

Actually, the concept of working backward isn't really new to us. When we're asked, "Who is the daughter of the neighbor of your brother?" what is the first question we must ask ourselves? Right: which brother? Then we need to identify his neighbor before we can determine the neighbor's daughter. In English, the "of the" prepositional phrase usually requires us to work backward.

Advanced Note: Assuming that we've already created the relationships shown in Figure 1-16 in the AutoLISP world, we can represent this problem as AutoLISP code in the following way:

```
Command: (cdr (car g))
7
```

When we have nested expressions we always execute the innermost parentheses first and work our way outward. This is how we do things backward in AutoLISP. The way we read this expression is, "Get the value of g (the top cons), find its car (the bottom cons), and return the cdr of *that* cons (the number 7)."

Background Note: AutoLISP users always wonder about the origin of the words *car* and *cdr.* After all, they aren't your everyday computer terms!

John McCarthy wrote Lisp on an IBM 704, a computer that had several fast-access index registers. A *register* is a memory location that is actually built into the CPU. Words in memory were placed in registers so that they could be accessed very quickly. Each word on the IBM 704 had an *address* part and a *decrement* part, which could be stored in different registers.

We can think of a cons as a memory location that contains the addresses of two other AutoLISP objects. Because conses are used so frequently in Lisp, McCarthy decided to store each of the cons' addresses in a register. The terms *car* and *cdr,* therefore, stand for ***C***ontents of ***A***ddress ***R***egister and ***C***ontents of ***D***ecrement ***R***egister, respectively.[1]

♦ ♦ ♦

Whereas a cons is an AutoLISP object, *car* and *cdr* express *relationships.*

As we have seen, every cons must have a car and a cdr. But there is no AutoLISP object called a car or a cdr; these terms merely express relationships between a cons and other AutoLISP objects.

In real life, if you see your friend Bob walking down the street with an uncle whom you don't know, you might turn to me and say, "Here comes Bob with another man." You wouldn't say, "Here comes Bob with an uncle." *Uncle* is a term that denotes that man's relationship with Bob. The person's "object type," so to speak, is *man;* his relationship to Bob is *uncle.*

The same man might also be your friend Dorothy's dentist. His object type is still *man,* but his relationship to Dorothy is *dentist.*

Look again at two conses shown earlier:

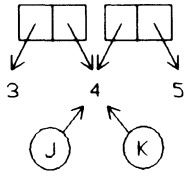


Figure 1-17

[1]John McCarthy, "History of LISP," *ACM SIGPLAN Notices,* Vol. 13, No. 8, p. 218, August, 1978.

The number 4 is the cdr of the first cons as well as the car of the second. This is perfectly valid. Its object type is *number,* but its relationship to the first cons is *cdr.* 4 is also the value of the symbols `j` and `k`; *value* is a relationship between a symbol and another AutoLISP object. It is important to keep the AutoLISP object types (numbers, symbols, strings, conses, and others that have not yet been presented) distinct from the relationships (values for symbols, cars and cdrs for conses).

Incidentally, object types such as numbers, symbols, and strings are called *atoms.* They are end products, not composed of other AutoLISP objects. Because a cons *is* made up of other objects (i.e., it's built out of its car and cdr), it is said to be *non-atomic.* Conses and lists (which we'll see next) are the only object types that are not considered atoms.

You might find it helpful to do Lab Exercises 4 to 6 before proceeding.

1.8 Lists

Lists are one of the most frequently used constructs in the AutoLISP language. Figure 1-18 shows a sample list:

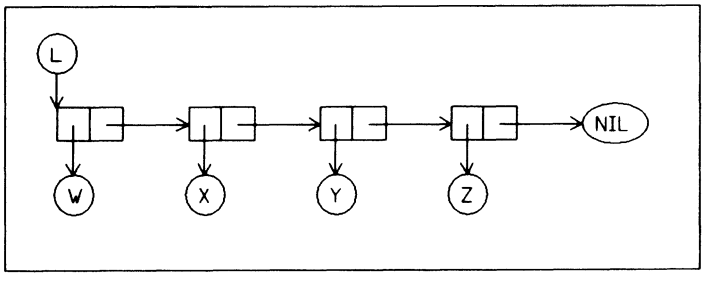


Figure 1-18

A list can be defined in three ways:

1. Assume we have a cons and, for the time being, we don't care what its car is. However, its cdr is another cons:

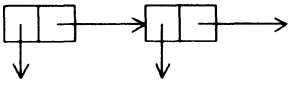


Figure 1-19

Next, assume that the cdr of that cons is yet another cons:

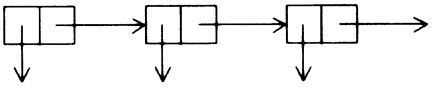


Figure 1-20

As you might imagine, we can string these conses together like a freight train. A train might have three boxcars or it might have 300 boxcars; so, too, we can connect as many conses as we choose. At some point we'll get to the end of the freight train (although it sometimes seems dubious when we're waiting at a railroad crossing!). Similarly, this collection of conses must also come to an end.

If the cdr of the very last cons is the symbol `nil`, then we have a structure called a *list:*

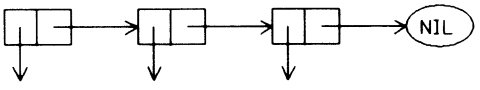


Figure 1-21

If the cdr of the last cons is anything but `nil`, then we do not have a list. Figure 1-22 shows a structure that looks rather like a list but isn't one because the cdr of the last cons is not `nil`.

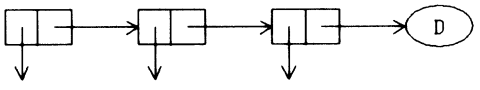


Figure 1-22

If the above picture is not a list, then what is it? Well, we don't actually have a name for it; it's just a collection of conses, linked by their cdrs, ending in `d`.

As we'll discover, AutoLISP makes it very easy to create a list but much more difficult to create the structure shown in Figure 1-22. As a result, we'll *never* accidentally create a structure that appears similar to a list, but isn't for reason that the cdr of the last cons is not `nil`.

2. A cons is a list if we can "cdr" it to `nil`. In other words, if we have a cons and continually take cdrs until we finally reach `nil`, then that cons is a list. This definition uses the word *cdr* as a verb.

If a cons is a list, then every cons we pass through as we cdr to `nil` is also a list. This means that if a cons is a list, then the cdr of that cons is also a list.

There is a third way of defining a list that is actually more precise, but it's a little tricky. It requires that we learn one more trait of the symbol `nil`, the weirdest one of all: `nil` is not only a symbol, `nil` is also defined to be a list!

`nil` is what's known as a "null list," an "empty list," or a "zero-element list." It's a list of nothing, but a list nonetheless. When AutoLISP needs for it to be a list, then it's a list. When it's needed to be a symbol, then it's a symbol. `nil` is the only object that does this "double duty," making it perhaps the strangest object in all of AutoLISP.

Although it initially may appear confusing, this aspect of `nil` is transparent to us once we begin coding. It's a necessary part of the language, however, because functions that sequence through lists return an error if they encounter an object other than a list (such as the symbol `d` in the "non-list" shown in Figure 1-22).

When list functions reach the end of the list they encounter `nil`, and they don't return an error because `nil` is itself a list. The most precise definition of a list, then, is as follows:

3. A list is either

   A. the symbol `nil` or
   B. a cons whose cdr is a list.

   This definition is a little bit of a brain-twister because it's recursive: a list is defined in terms of itself. An example should clarify the definition.

Example

In Figure 1-23 is `nil` a list? Why or why not?

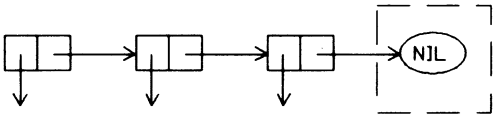


Figure 1-23

`nil` is a list because it fulfills rule A above; this is an axiom of the language. Is the cons to `nil`'s left a list? Why or why not?

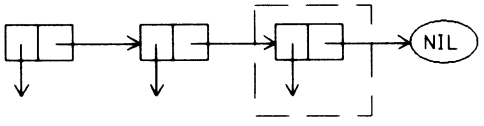


Figure 1-24

This cons is a list because its cdr is `nil`, and `nil` is a list. Rule B states that a cons is a list if its cdr is a list. Is the next cons a list? Why or why not?

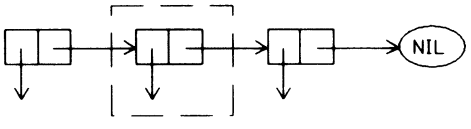


Figure 1-25

This cons is a list because its cdr is the cons to its right, which we just determined is a list. Again, a cons is a list if its cdr is a list. How about the first cons?

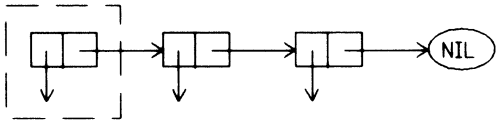


Figure 1-26

This cons is also a list for the reasons just stated. In fact, each one of these conses is a list. This complies with the second definition we gave of lists: If a cons is a list, then every cons we pass through as we cdr to `nil` is also a list.

The last definition is the nicest because it lays to rest a common misconception about lists. When discussing lists, we are often sloppy and refer to the entire structure as a list. But AutoLISP doesn't see it that way; it treats *each individual cons* as a list.

When we call a function that operates on lists, we can supply this list to the function and it will run:

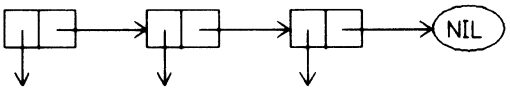


Figure 1-27

However, we can also supply this list to the function and it will run just as well:

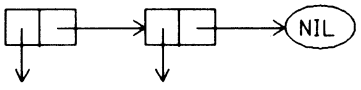


Figure 1-28

In fact, we can supply, this list or even `nil` and the list function will run correctly:

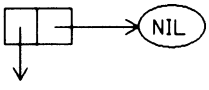


Figure 1-29

In later chapters we will encounter functions that manipulate lists of data. We are not compelled to supply these functions with the "entire list"; it's sometimes necessary that we perform an operation on a portion of the list. It is important to recognize that although we often refer to the entire structure as a list, AutoLISP treats each individual cons as a list.

We have now examined what goes into the making of a list and seen how AutoLISP uses the cdrs of the conses to support the list structure. But what about the *cars* of the conses?

We build lists to collect information that is somehow related. We might have a list of symbols, numbers, 2-by-4s, electrodes, or a collection of different types of information. The cars of the conses that form a list are the objects that we are collecting; they are called the *elements* or *members* of the list. These two terms mean the exact same thing and are used interchangeably.

In Figure 1-18, at the start of this section, `w`, `x`, `y` and `z` are the members of the list that is the value of the symbol `l`.

Figure 1-30 shows a list whose members are different object types:

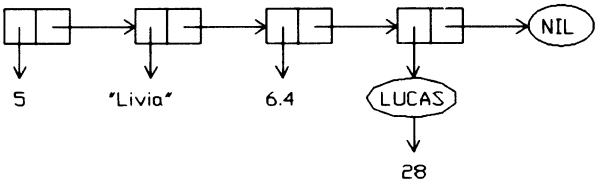


Figure 1-30

We use the term *backbone* to describe the conses that are used in the construction of a list, that is, the conses that we pass through as we cdr to `nil`. This name is apropos because each cons looks like a vertebra, and because the purpose of these conses is to support or "hold up" the list in the same way that our backbone supports our bodies.

Recall that *any* AutoLISP object can be the car of a cons, including another cons. This implies that elements of lists can themselves be lists, which means that lists can be nested many levels deep. Figure 1-31 shows a list nested two levels deep:

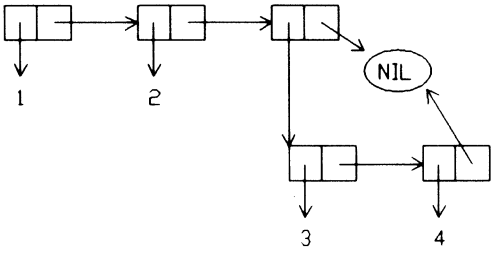


Figure 1-31

Figure 1-32 shows a list nested three levels deep:

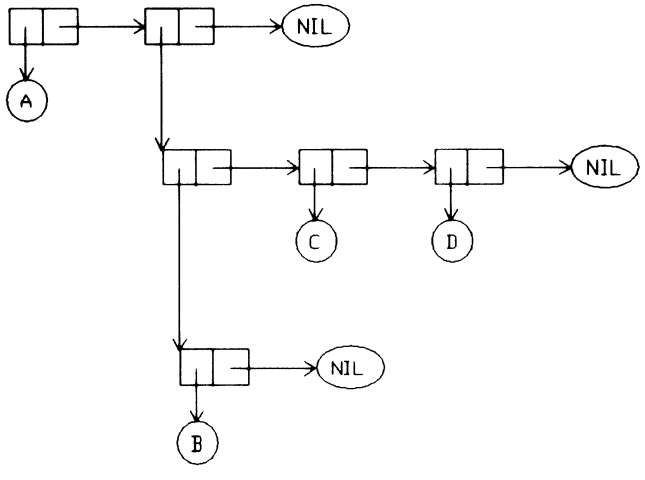


Figure 1-32

In this picture, the top-level list has two conses in its backbone; thus, we say that the *length* of this list is two. As will become apparent when we study

AutoLISP code, it is quite important that we be able to examine a list and determine its length.

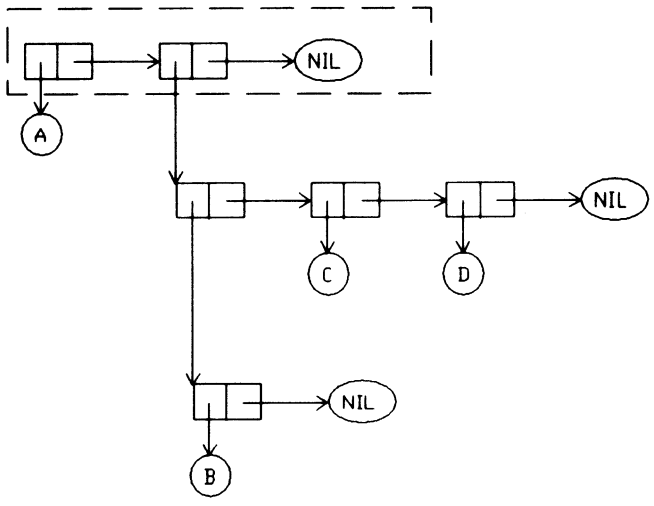


Figure 1-33

The top-level list has two members or elements, a and another list. Since the members of a list are the cars of the backbone conses, a list of length two has two members as well. However, since the members can themselves be lists, it is usually easier to determine a list's length by counting the conses in the backbone rather than counting the number of elements.

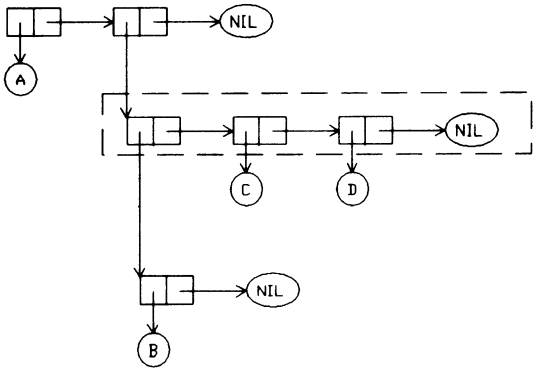


Figure 1-34

The second member of the top-level list is another list. Since the second-level list has three conses in its backbone, its length is three. This means that it has three elements as well: another list, c, and d.

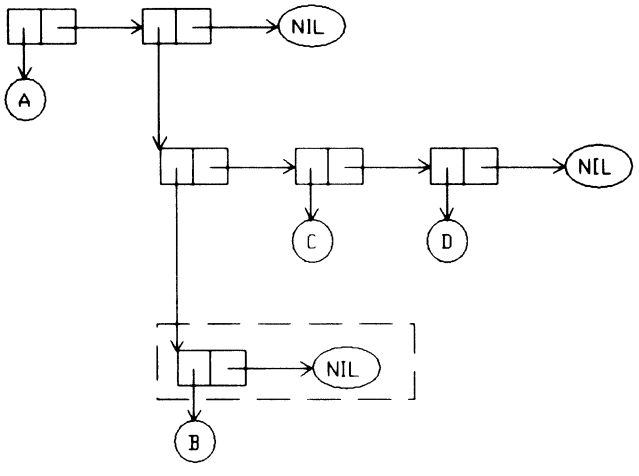


Figure 1-35

The first element of the second-level list is itself a list of length one. This third-level list has one cons in its backbone and one element, `b`.

You might have noticed that in Figure 1-31 the two lists end with arrows pointing to the same symbol `nil`, but in Figure 1-32 each list ends with its own `nil`. In AutoLISP, there is only one symbol `nil`, and to make the pictures be absolutely correct we should draw our lists as we did in Figure 1-31.

When we draw multitiered lists, it is inconvenient and even confusing to draw arrows all over the page just to reach the single symbol `nil`. Instead of explicitly drawing `nil`, we use a modified grounding symbol to end our lists:

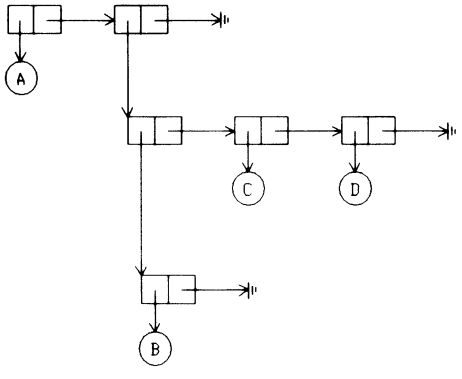


Figure 1-36

The thought is that the list is "grounded" into the page and attaches to the one symbol `nil` that exists somewhere in the AutoLISP world. This may initially sound strange, but it works well and is quite convenient. We'll almost always end our lists with this grounding symbol, but keep in mind that it is merely a synonym for `nil`.

You might find it helpful to do Lab Exercises 7 to 10 before proceeding.

Specifying Points

In AutoLISP, a 2D point is expressed as a list of two real numbers:

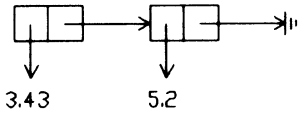


Figure 1-37

The X coordinate is the car of the list, 3.43; the Y coordinate is the car of the cdr (*not the cdr!*) of the list, 5.2. Two arrows are used to connect the first cons to the second list element, so it takes two steps to get there.

Likewise, a 3D point is represented as a list of three real numbers, corresponding to the X-Y-Z axis:

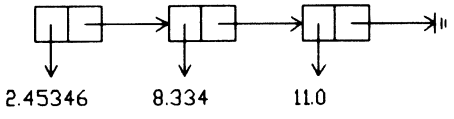


Figure 1-38

The X coordinate, 2.45346, is the car of the list. The Y coordinate, 8.334, is the car of the cdr. The Z coordinate, 11.0, is the car of the cdr of the cdr.

Although points are lists of real numbers, we can supply integers and AutoLISP will convert them to reals.

1.9 Functions

Functions are AutoLISP objects that we use to accomplish a task. We call upon a function to perform an action, to return some information about the AutoLISP world, and/or to actually effect a change in the AutoLISP world.

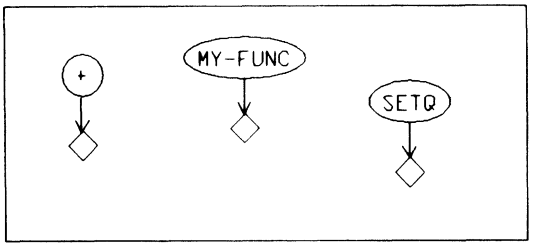


Figure 1-39

We represent a function as a diamond box, which tells us nothing about what the function does. This is because we cannot examine a built-in AutoLISP function or see its code. One function pretty much looks like any other. Although this may appear to handicap us, once we become more involved with the actual coding we'll see that it doesn't cause any problems.

We say that we *run, call,* or *invoke* a function. *All* functions perform the following steps:

- Functions request zero or more *arguments.* An argument can be any AutoLISP object, depending on the given function. It is the data that the function uses when it runs.

- Functions make sure that they have the *right number* of arguments. Most functions require a specific number of arguments. If a function expects two arguments and we supply three or more, we get a "too many arguments" error. If we provide zero or one argument, the function issues a "too few arguments" error. Or perhaps it just issues the error, "incorrect number of arguments." All of these messages mean the same thing: We didn't supply the number of arguments the function needs to run.

- Functions make sure that they have the *right type* of arguments. The addition function, for example, adds numbers together; it requires numbers as arguments. If we give the addition function symbols, strings, lists, or any type of object other than numbers, it is unable to perform its task. Instead, it issues a "bad argument type" error and aborts the process.

- If the function receives the right number and right type of arguments, then it performs an action. For example, the addition function adds its arguments.

- When it is done running, the function returns a result (unless we did something wrong, in which case it signals an error instead).

Every function returns a result. If the function was called to gather information, then the result is that information. If the function was called to perform an action, it still returns a result that might be used as input to another function.

We call this set of steps the function's *contract.* Every function has a contract (or, we sometimes say, we have a contract with each function that we run). For a function to execute properly, we must fulfill our end of the contract, which is to supply the right number and right type of arguments. If the function discovers that we have broken the contract, it immediately aborts and returns an error message. But if we fulfill our end of the contract, the function will complete its end, which is to execute the action that we requested and to return the result that we expect.

It is not sufficient to learn what each AutoLISP function does; we must study its complete contract. Running a function without that knowledge is like trying to bake a cake with a recipe that doesn't specify the amount of each ingredient! Unless we know the number and type of arguments that the function accepts, we will continually get the same errors and spend hours of unproductive time debugging. For this reason, whenever a function is presented in this book, we will always examine its full contract.

Advanced Note: We provide access to a function by making it be the value of a symbol. Thus, a symbol can name a function as well as a number, another symbol, a cons, and so on. This is a very important difference between AutoLISP and other programming languages. For example, in BASIC or FORTRAN, + is a *function.* In AutoLISP, + is a *symbol* whose value is the addition function:

Figure 1-40

As with any symbol, the value of + can be changed . . . usually with disastrous results. For example, we might accidentally enter the following code:

```
Command: (setq + 3)
3
```

Figure 1-41

We have now redefined the + symbol and can no longer perform addition. Don't do this! To recover, simply save and reopen the drawing. A clean version of AutoLISP is loaded whenever we open a new drawing.

♦ ♦ ♦

In Section 5.1 we learn how to create our own functions. For all practical purposes, our own functions are indistinguishable from the built-in AutoLISP functions. If we write a function and call it `test`, it sits side by side with the multiplication and addition functions:

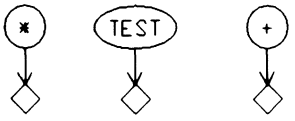


Figure 1-42

In its implementation of AutoLISP, Autodesk has chosen to distinguish built-in functions from those that we write ourselves. Functions that have been defined in the AutoLISP language itself are called *subrs*.

We will encounter several instances where AutoLISP treats subrs differently from functions that we write. These differences are minor, and for the most part we can assume that all functions work in the same manner, regardless of whether they were written by us or by Autodesk.

A large part of learning AutoLISP is studying the subrs that are built into the language. The next section introduces some basic ones.

Note: It is good programming practice to *comment* functions that we write ourselves. In AutoLISP, a semicolon is a comment character. On any line of code AutoLISP ignores everything to the right of a semicolon. For example:

```
Command: (+ 12 14)          ;This adds 12 to 14
26
```

Commenting style is discussed in Section 13.1.

1.10 Basic Functions

Nearly 200 subrs have been defined in AutoLISP for our use. Learning their contracts is analagous to learning the many AutoCAD commands. The more

that we know, the larger our "toolbox," and the more flexible and powerful our programming becomes.

As with AutoCAD commands, it's not necessary that we learn all of the functions at once. Most of us learned the LINE, MOVE, and ZOOM commands before the 3DFACE and HATCH commands, and we use them more frequently as well.

My aim is to get the essential and most commonly used AutoLISP subrs into our toolboxes before the more esoteric ones are presented. Although we must know a certain threshold of functions before we can become productive, there are many functions whose contracts should be learned only when they are needed. Wherever possible, *Essential AutoLISP* indicates which functions are absolutely necessary to learn immediately and which ones can safely be put off until a later time.

The first time you read this book, try to become familiar with all the functions' contracts so that you have an idea of what tools are available for your use. When you need to use a particular function, go back and study its contract in depth.

The subrs described in this section are the most commonly used, low-level AutoLISP functions. Other AutoLISP functions use these functions to accomplish their work, and we'll use most of them quite often as well.

The functions are defined in this section in a manner used throughout this book. First, the name of the function is shown in code font. It is followed by the number and type of arguments that the function takes (in normal type) and a statement of the function's contract. Remember that any AutoLISP object can be an argument to a function, and it is up to the individual function to determine what object types its arguments can be. For example, `setq` is defined as follows:

`setq` sym obj	Takes a symbol and any AutoLISP object and makes the object be the value of that symbol. It returns the object.

We can tell that `setq` takes two arguments because there are two words next to its name. *sym* indicates that the first argument must be a symbol. *obj* means that the second argument can be any AutoLISP object that the user chooses to provide.

`list` obj . . . obj	Takes any number of AutoLISP objects as arguments and creates a list with each of these objects as members. It returns the new list.

`list` is the most general AutoLISP function in that it takes as many arguments as we wish to give it (as indicated by the ellipses [. . .]), and they can be any AutoLISP objects that we want.

Following the function name and arguments is a statement of the function's contract. To use any AutoLISP function correctly, we must know its name, the number and type of arguments it takes, what it does, and what it returns.

Appendix F contains a list of all of the functions and their contracts, as well as an explanation of the argument abbreviations used.

With the understanding that you will be coming back to this section after you have read subsequent chapters, we include not only the contract of the function, but also examples of how each function is used. *On the first reading, you don't have to concern yourself with how the function is written or the results that are returned.* These will be explained in later chapters.

If you'd like to try the examples, you can enter them at the Command: prompt. Be sure that you type them in exactly as shown, and that the parentheses and double quotes balance. If you make a mistake or get an error message, simply press `<esc>` and the Command: prompt will return. Here's an important point to keep in mind:

> Nothing we do in AutoLISP will harm our drawing, AutoCAD, or AutoLISP itself!

The *worst* thing that can happen is we'll have to exit and reenter the drawing. But even that situation arises very rarely.

`setq` sym obj	Takes a symbol and any AutoLISP object and makes the object be the value of that symbol. It returns the object.

`setq` is the assignment function in AutoLISP. It is used to give a value to a symbol. In other computer languages, = is used to assign a value to a variable.

In AutoLISP, = has another application, so we use `setq` instead. For example, in BASIC we might enter

```
x = 20
```

The same assignment is performed in AutoLISP by entering

```
Command: (setq x 20)
20
```

It creates the following relationship between x and 20:

Figure 1-43

Once a symbol has a value, we can use `setq` to change that value:

```
Command: (setq x "Pump")
"Pump"
```

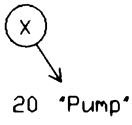


Figure 1-44

The 20 doesn't go away, but it's no longer the value of x. Notice that the first argument to `setq` *must* be a symbol. If it's not, `setq` returns a "bad argument type" error.

We can learn the value of a symbol by typing an exclamation point followed by the symbol's name at the Command: prompt:

```
Command: !x
"Pump"
```

`setq` is actually a fairly complex function; there's more to it than is immediately apparent. The complete definition of `setq` is presented in Section 5.1.3.

`cons` obj obj	Takes two AutoLISP objects as arguments and creates a new cons with the first object as its car and the second object as its cdr. It returns the new cons.

`cons` is the function that creates conses, the basic building block of AutoLISP structures. Other functions, such as `list`, appear to create conses, but they actually call the `cons` function to perform the "consing." A cons whose car is 2 and whose cdr is 4 can be created as follows:

```
Command: (cons 2 4)
(2 . 4)
```

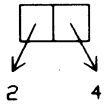


Figure 1-45

The dot between the car and the cdr is part of the printed representation of the cons. It's explained in Section 2.2.

Although this code creates a new cons, we have no way of accessing it. This is because conses don't have names. Having been created, this cons will float around the AutoLISP world until the "garbage collector" sweeps it up and discards it.

The usual method of providing access to an unnamed object is to make it be the value of a symbol. To accomplish this we put the `cons` function inside of a `setq`:

```
Command: (setq c (cons 2 4))
(2 . 4)
```

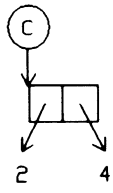


Figure 1-46

Let's run the cons function with 2 and 4 another time:

```
Command: (setq d (cons 2 4))
(2 . 4)
```

Each time we call the cons function it builds and returns a brand new cons. We can draw these relationships as follows:

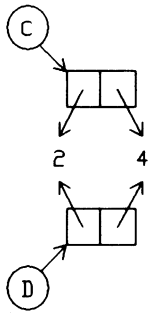


Figure 1-47

Note that there is only one number 2 and one number 4 in the AutoLISP world. Even though a new cons is created, the numbers are reused. Symbols and strings are reused too. Although it might seem easier to reuse an already existing cons, if several conses have 2 and 4 as their respective cars and cdrs, the cons function wouldn't know which one to return.

The cons function takes exactly two arguments. If given only one, it returns a "too few arguments" error. By the same token, if we give it three or more arguments, we get back a "too many arguments" error.

Autodesk has implemented the entity database as a list of conses. Therefore, it is extremely important that we understand conses' properties in order that we may manipulate the entity database correctly.

Any AutoLISP object can be the car or cdr of a cons, including another cons. The following structure is entirely permissible:

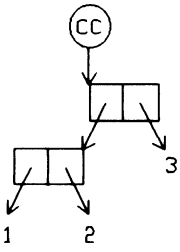


Figure 1-48

The cons that is the value of the symbol `cc` has a cons as its car. The following code creates this structure:

```
Command: (setq cc (cons (cons 1 2) 3))
((1 . 2) 3)
```

Aside: We know that in AutoCAD there is an entity called a line, and that it's created by a command named LINE. Similarly, in AutoLISP, the cons object is created by the `cons` function. Furthermore, as we'll see shortly, a cons' car and cdr can be accessed by the `car` and `cdr` functions, respectively. These words do double duty, but to avoid confusion they are always printed in code font when representing a function's name.

`list` obj . . . obj	Takes any number of AutoLISP objects as arguments and creates a list with each of these objects as members. It returns the new list.

In terms of its arguments, `list` is the most general function in AutoLISP. It takes as many arguments as we'd like to give to it, and these arguments can be any datatypes we choose. If we supply the `list` function with 3 arguments, it will build a list of 3 elements; if we provide 30 arguments, the list will have 30 elements, and so forth.

The following examples show its usage:

```
Command: (setq list1 (list 1 2 3 4))
(1 2 3 4)
```

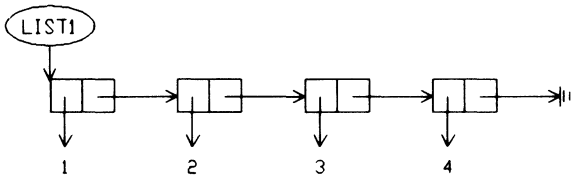


Figure 1-49

```
Command: (setq list2 (list 1 "Wall" 2 "Ceiling" 3 "Floor"))
(1 "Wall" 2 "Ceiling" 3 "Floor")
```

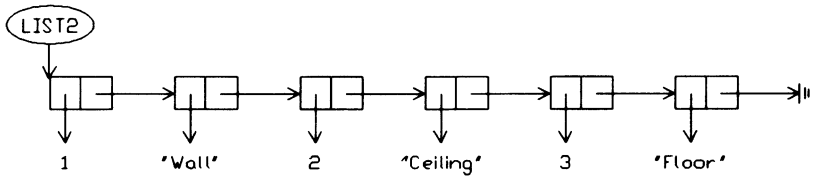


Figure 1-50

Since lists are built out of conses, they don't have names. We must remember to bind a symbol to the list (with `setq`) or provide another way of accessing it, or it will be unrecoverable. But here's a little "gotcha": One name we *cannot* call our lists is `list`. Why? Because that's the name of this subr. If we call a list `list`, we'll rebind and lose access to the `list` function!

We can use the `list` function to build nested lists, as follows:

```
Command: (setq list3 (list 1 2 (list 3.1 3.2) 4))
(1 2 (3.1 3.2) 4)
```

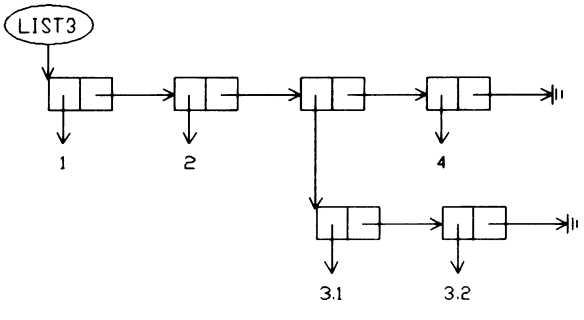


Figure 1-51

`car` cons	Takes a cons as its only argument and returns the car of that cons.
`cdr` cons	Takes a cons as its only argument and returns the cdr of that cons.

Unlike the `cons` and `setq` functions, which change the AutoLISP world, `car` and `cdr` are information-gathering functions only. They enable us to identify the car or cdr of any cons, or find an AutoLISP object buried in a cons structure.

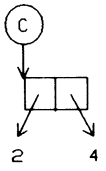


Figure 1-52

When we give the `car` function this cons, it returns 2. Given the same cons, the `cdr` function returns 4.

```
Command: (car c)
2

Command: (cdr c)
4
```

`car` and `cdr` can be used with more complex structures, such as the value of `x`:

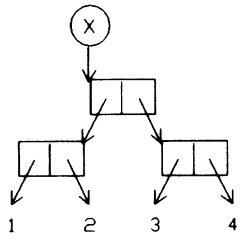


Figure 1-53

```
Command: (cdr x)
(3 . 4)
```

These functions can be used jointly to probe deeper into AutoLISP structure:

```
Command: (car (car x))                ;1 is the car of the car of x
1

Command: (cdr (car x))                ;2 is the cdr of the car of x
2

Command: (car (cdr x))                ;3 is the car of the cdr of x
3

Command: (cdr (cdr x))                ;4 is the cdr of the cdr of x
4
```

Our most important early task in the study of AutoLISP is to gain an understanding of how to sequence through cons structure via cars and cdrs. This is

a first example; many more will be shown throughout this book. If you're new to AutoLISP, don't worry if you don't totally understand the code; it will be explained in later chapters.

Q: What is the relationship of the symbol `v` to the following list? In other words, what function do we use to access `v` ?

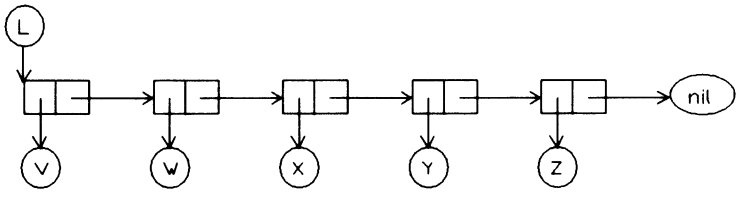


Figure 1-54

A: `v` is the *car* of the list. We use the `car` function to access `v`.

Q: What is the relationship of the second cons to the entire list?
A: The second cons is the *cdr* of the list. We use the `cdr` function to access the second cons.

Q: What is the relationship of the symbol `w` to the list? How do we access `w`?
A: `w` is the *car of the cdr* of the list. We access `w` as follows:

```
Command: (car (cdr l))
W
```

Most people find it easier to start with the object they want (e.g., `w`) and work backward (as usual, everything in AutoLISP is backward!). `w` is the *car* of the cons above it, which is the *cdr* of the cons to its left; that cons is the *value* of the symbol `l`. Thus `w` is the car of the cdr of the value of `l`. We can specify these relationships pictorially as follows:

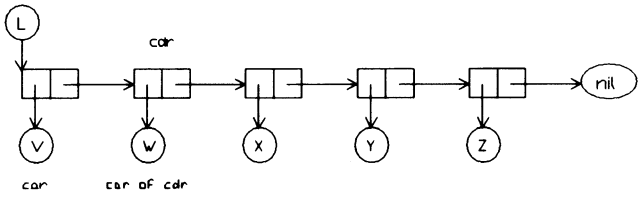


Figure 1-55

Complete this picture by specifying the sequence of cars and cdrs needed to access each cons and each element in the list. The solution follows.

Solution

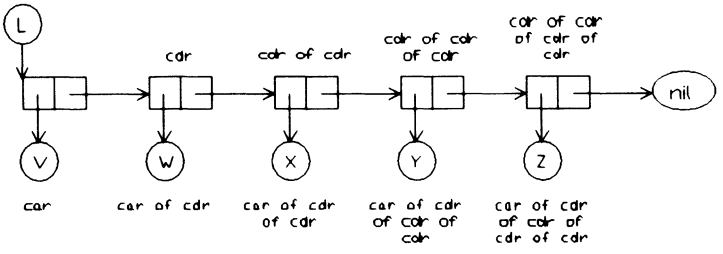


Figure 1-56

```
Command: (cdr (cdr l))
(X Y Z)

Command: (car (cdr (cdr l)))
X

Command: (cdr (cdr (cdr l)))
(Y Z)

Command: (car (cdr (cdr (cdr l))))
Y

Command: (cdr (cdr (cdr (cdr l))))
(Z)

Command: (car (cdr (cdr (cdr (cdr l)))))
Z

Command: (cdr (cdr (cdr (cdr (cdr l)))))
nil
```

Here are some very important observations about this list:

- If we have a list and take its cdr, we get a list. The cdr of the five-element list (V W X Y Z) is the four-element list (W X Y Z). If we "cdr" this list three times, we get a two-element list, (Y Z). No matter how many times we cdr a list, we *always* get a list, *never* an element of the list. If we cdr past the end of the list, we get `nil`.

- If we cdr a list zero or more times and then take a car, we get an element of the list. Elements are always *cars* of lists (after all, that's the definition of an element!). As we have seen, the car of a list could be another list.

Common Lisp has two functions, `first` and `rest`, that are equivalent to `car` and `cdr` when used with lists. The car is the *first* element of any list we create. The cdr is the *rest* of the list if we remove the car. These functions are excellent mnemonics for remembering how to sequence through lists.

As we can see, sequencing through even a simple five-element list requires a considerable number of `car` and `cdr` functions. To make our lives simpler (and cut down on our typing), AutoLISP has several subrs that are used for abbreviating the combinations of `cars` and `cdrs`. The basic ones are `cadr` and `cddr`.

`cadr` cons	Takes a cons as its only argument and returns the car of the cdr of that cons.
`cddr` cons	Takes a cons as its only argument and returns the cdr of the cdr of that cons.

We use the `cadr` function quite frequently, but other abbreviations less often. If you're just beginning to learn AutoLISP, you might consider coming back to this explanation a little later. Following are the rules for abbreviating:

1. The abbreviation functions always begin with the letter *c* and end with the letter *r*. Since `car` and `cdr` both begin with *c* and end with *r*, these functions just follow suit.

2. Between the *c* and *r* we specify any combination of *a*'s and *d*'s such that each *a* stands for a *car* and each *d* stands for a *cdr*. They should be in the same order as the cars and cdrs in the long version.

3. There can be no more than four *total a*'s and *d*'s in any one abbreviation.

Now it's your turn. Here's the list we've been using with the preceding abbreviations filled in. Enter the abbreviations for the remaining list elements and conses (accessing `z` is tricky) before examining the solution that follows.

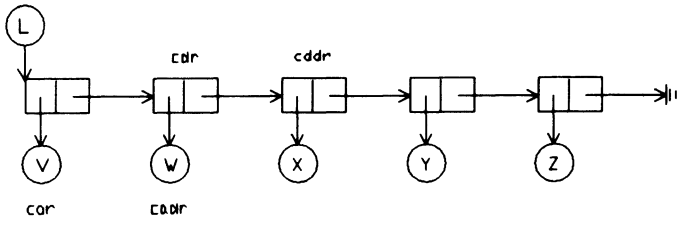


Figure 1-57

Solution

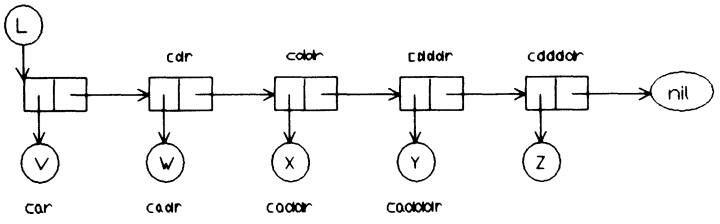


Figure 1-58

Most people get to this point without difficulty, but some have trouble specifying the abbreviation for accessing the last element, `z`. `caddddr` is incorrect because that takes five *a*'s and *d*'s, and our limit is four.

When we need to access an object that is more than four arrows away, we must use more than one function. We can access the symbol `z` in any one of the following ways:

```
(car (cddddr l))     1 a, 4 d's
(cadr (cdddr l))     2 a's and d's, 3 d's
(caddr (cddr l))     3 a's and d's, 2 d's
(cadddr (cdr l))     4 a's and d's, 1 d
```

Rather than putting five letters in between the *c* and *r*, we split it into two functions. To access `nil` in this list we might enter `(cdr (cddddr l))`.

Keep in mind that `z` and `nil` can still be accessed the long way; the abbreviations are not required. Since we commonly sequence through lists of atoms, these abbreviations are the ones we most frequently use. However, any combination of *a*'s and *d*'s is permissible. Here's a list we saw a few pages back:

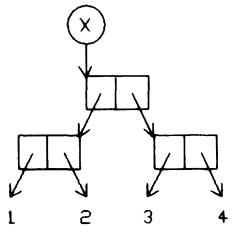


Figure 1-59

The following functions access various components of the list; the abbreviations are shown in the comment fields. Notice that in each of these abbreviations the *a*'s and *d*'s are in the same order as in the longer version.

```
Command: (car (car x))          ;(caar x)
1

Command: (cdr (car x))          ;(cdar x)
2

Command: (car (cdr x))          ;(cadr x)
3

Command: (cdr (cdr x))          ;(cddr x)
4
```

Other permutations, such as `cdaar`, `cadadr`, and `cdddar`, are valid but less commonly used. All of these combinations are individually defined functions in the language, meaning that there are 16 subrs for sequencing through list structure in this manner.

Important Notes

- The most important information in this chapter concerns the construction of lists and how to sequence through them using the `car` and `cdr` functions. We use these functions constantly, not only to manipulate lists of data, but also when we work with code. Understanding the relationships depicted in Figure 1-56 will be crucial once you get to Chapter 4.

- The abbreviations we have just seen are not really that important. If you find them confusing, *don't use them!* What good is a shortcut that takes longer than the long version? After you play around with AutoLISP for a short while these abbreviations will make more sense, and you can add them to your toolbox. For now, put the various `car` and `cdr` functions together the long way.

- At the risk of being contradictory, there are a couple of abbreviations that we *must* get to know, by rote if necessary. These are `cadr` and `caddr`. The `cadr` function accesses the second element of a list. When working with points, the Y coordinate is the cadr (*not* the cdr!). Similarly, the `caddr` function accesses the third element of a list, which is the Z coordinate of a point. The X coordinate, of course, is the car.

`print` obj	Takes an AutoLISP object as its only argument and prints it on the terminal screen. It returns that object.

`print` is the most commonly used function for printing AutoLISP objects. It is one of several functions, including `princ`, `prin1`, `prompt`, and `write-line`,

that print an AutoLISP object on the terminal screen. `print` and its variations are explained in detail in Section 8.2.

In most instances we want a function that we write to *return* a value (so we can use it elsewhere) rather than *print* it, because AutoLISP automatically prints our results for us. We'll expand on this concept later in the book. For now, use `print` when you want to print information on the screen.

`load` str	Takes the name of an AutoLISP file and loads that file into the AutoLISP world. It returns the name of the last function in the file.

Suppose we've entered one or more functions using an editor and saved them away in the file `test.lsp`. To run these functions we must load this file into memory, as follows:

```
Command: (load "test")
"test"
```

If the file is not in our current working directory, we can load it by specifying a complete pathname:

```
Command: (load "c:/acad/project/test")
"test"
```

Notes

- `load` assumes a `.lsp` extension; we don't need to include it. Our DOS AutoLISP files should always have a `.lsp` extension. If we want to load an AutoLISP file with a different extension, we must supply both the filename and extension to `load`.

- New users are often surprised to see the path components separated by a forward slash instead of a backslash. This is because backslash has a special meaning in AutoLISP. However, those backslash fans among us can use a *double backslash* to separate path components, like so:

```
Command: (load "c:\\acad\\project\\test")
"test"
```

- If we use an editor to modify a file that we've already loaded, we must reload that file for the changes to take effect. In other words, after the editor

writes our altered file to the disk we must load the current copy of the file so that AutoLISP knows about these changes.

- When we load a file we might get one of several error messages:

"malformed list"	A function is missing a close parenthesis somewhere.
"extra right paren"	There are too many close parentheses. Be aware that we might get this message if we're missing an *open* parenthesis somewhere.
"LOAD failed"	`load` can't locate the file. If we're certain that the file exists, we should verify that we've spelled its name correctly, that we've specified the correct path, and that we gave the file a `.lsp` extension.
"malformed string"	We forgot the double quotes to end a string. This error sometimes returns an "exceeded maximum string length" message.

Advanced aspects of `load` are presented in Section 8.3.1.

1.11 Important Points to Remember

- There are two kinds of numbers, *integers* (counting numbers) and *floating point* or *reals* (decimal numbers). Real numbers that are less than 1 must have a leading 0.

- Every symbol *must* have a value, which can be any AutoLISP object type. We say that the symbol is *bound* to that object. If we do not explicitly provide a value (usually via `setq`), then a value of `nil` is automatically assigned to the symbol. In that case the symbol is said to be *unbound* and the relationship is not shown in our drawings. Remember that a symbol which appears to have no value is actually bound to `nil`.

- Never change the value of the variables `pi` or `t`.

- Never *backtrace* arrows. Always move in the direction in which an arrow points.

- Never *trace through* arrows. The value of a symbol is the object the symbol's arrow points to . . . even if that object has an arrow pointing to yet another

object. If the car of a cons is another cons, don't trace through the latter cons' arrows to get to an atom.

Every cons *must* have both a car and a cdr; there are always two arrows emanating from a cons. The car and cdr can be any AutoLISP object, including another cons or a list. Every time we call the `cons` function, AutoLISP creates a brand new cons, even if it is identical to an already existing one.

- A cons is a list if and only if we can "cdr" it to `nil`. Each cons we pass through as we cdr to `nil` is also a list.

- The car of each cons in the backbone of a list is called a *member* or *element* of that list. Any member of a list can itself be a list.

- The members of a nested list are *not* members of the top-level list. A list has only as many members as there are conses in its backbone. The `nil` that ends the list is not a member of the list either; it's a cdr, not a car, of a cons.

- Although we often refer to the entire structure as a list, AutoLISP treats each individual cons in the backbone as a list. The number of conses in the backbone determines the length of the list. We don't include `nil` in this calculation.

- Points are lists of real numbers. The X, Y, and Z coordinates are the car, cadr, and caddr, respectively, of the point; the Y coordinate is *never* a cdr of a list. We can specify coordinates as integers and AutoCAD will convert them to reals.

- Functions are drawn using a diamond box because we don't ever see the code that constitutes subrs (built-in functions). A function, such as addition or the function that builds lists, is always the value of a symbol, such as + or `list`. When we give AutoLISP the symbol, it finds the symbol's value and uses the function.

- *Arguments* are the data that we supply to a function. If we supply the wrong number of arguments, the function indicates an error: "too many arguments", "too few arguments", or "incorrect number of arguments". If we supply the wrong kind of argument, the function issues a "bad argument type" error.

- AutoLISP is reloaded every time we open a new drawing, and nothing we do in AutoLISP can irreparably harm our drawing, AutoCAD, or AutoLISP itself. Therefore, never hesitate to try something new.

1.12 Labs

1. Which of the following are *not* valid numbers in AutoLISP? Why?

   a) –5

   b) 2S

   c) 6.6

   d) 3.4e8

   e) 3.004e-2

   f) "654"

   g) .2

   h) 3/4

   i) 3,4

   j) 4.123456789

2. Which of the following are not valid symbol names? Why?

   a) `56`

   b) `'xyz`

   c) `setq`

   d) `(x y)`

   e) `*`

   f) `*%$`

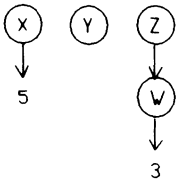
   g) `run-code`

   h) `c.34890a`

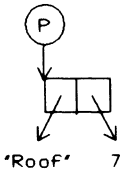
   i) `move`

   j) `long symbol name`

3. What are the values of the symbols x, y, z, and w?



4. Answer questions about the following picture:

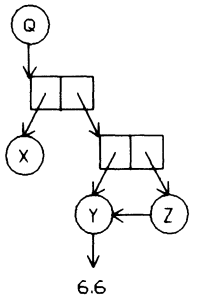


   a) What is the value of the symbol p ?

   b) What is the cdr of the value of p ? What kind of AutoLISP object is it?

   c) What is the car of the value of p ? What kind of AutoLISP object is it?

5. Answer questions about the following picture:



   a) What is the value of the symbol q ?

   b) What is the car of the value of q ? What kind of AutoLISP object is it?

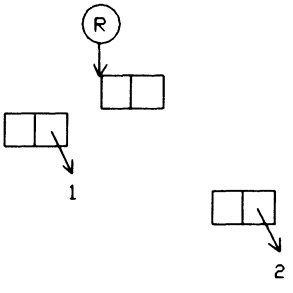
   c) What is the cdr of the value of q ? What kind of AutoLISP object is it?

   d) What are the values of the symbols x, y, and z ?

e) What is the car of the cdr of the value of `q` ?

f) In terms of cars and cdrs, what is the relationship between the symbol `q` and the symbol `z` ?

6. The following picture is incomplete. Fill it in according to these instructions:



a) For clarity, let's call the value of the symbol `r` the "first cons." Make the car of the first cons be the cons whose cdr is 1.

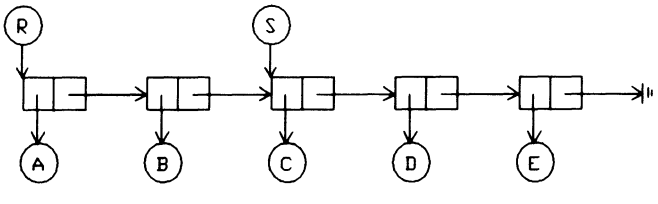
b) Make the cdr of the first cons be the cons whose cdr is 2.

c) Make the car of the cdr of the first cons be the symbol `s`.

d) Make the car of the car of the first cons be the symbol `u`.

e) Make the value of the car of the cdr of the first cons be the cdr of the cdr of the first cons.

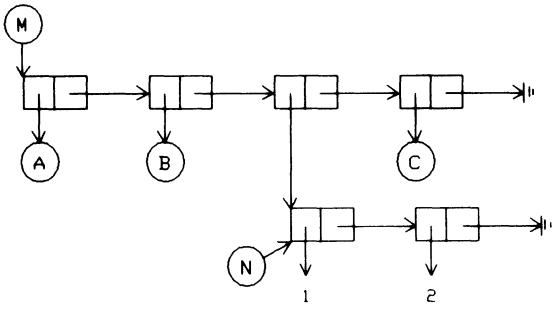
7. Answer the questions below about the following picture:



a) Is the value of `r` a list? What are its elements?

b) Is the value of `s` a list? What are its elements?

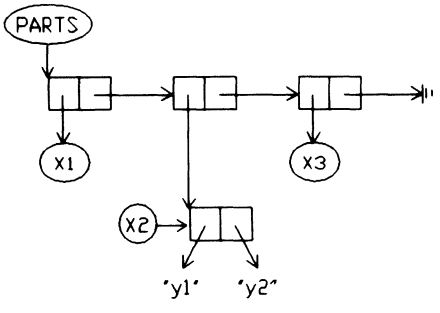
8. Answer the questions below about the following picture:



a) Is the value of `m` a list? What are its elements?

b) Is the value of `n` a list? What are its elements?

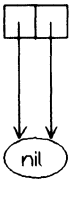
9. Answer the questions below about the following picture:



a) Is the value of `parts` a list? What are its elements?

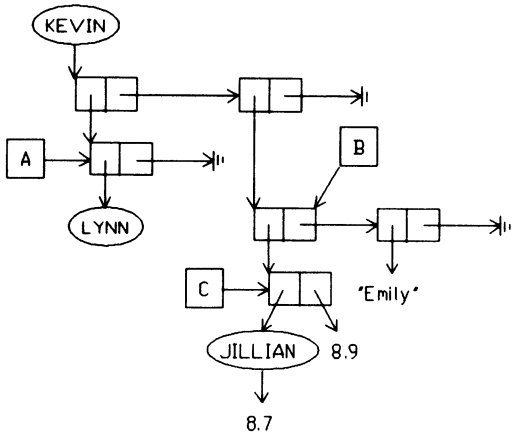
b) Is the value of `x2` a list? What are its elements?

10. Answer the questions below about the following picture:



a) Is this cons a list? What are its elements?

11. Prior to doing this exercise, be certain you understand the relationships shown in Figure 1-56. Then answer questions about the following picture.



a) What are the cars of conses labeled A, B, and C?

b) What are the cdrs of each of these conses?

c) What are the values of `jillian`, `lynn`, and `"Emily"`?

d) How many elements are in the list that is the value of the symbol `kevin`? What are they?

e) What sequence of cars and cdrs would we use to access the symbol `lynn`? In other words, what is `lynn`'s relationship to `kevin` ? If you have experience with AutoLISP, write the code that would return `lynn`.

f) Repeat Exercise *e* for `jillian`, 8.9, and `"Emily"`. Describe their relationship to `Kevin` and, if you can, write the code that will access them.

12. a) Draw the symbol `ingrid` and make its value be a list of length two.

b) Make the first element of this list be a cons with a car of `julie` and a cdr of `lisa`. Make the value of `lisa` be 23.

c) Make the second element be a list. The elements of this second-level list should be `bob`, `helen`, and another list.

d) The inner list should comprise the symbols `gary` and `ellen`. Make `ellen`'s value be the string `"Goldie"`.

2

Printed Representation of AutoLISP Objects

Topics Covered in This Chapter

- How the various AutoLISP object types are printed on the screen
- Conversion from printed representation to picture notation
- Conversion from picture notation to printed representation
- Background topic: Simplifying the printed representation of lists
- Prefix notation
- The DRAW on-line tutorial

Goals for This Chapter

- Recognize each AutoLISP object by its printed representation as well as by its pictorial representation.
- Learn how to convert a picture into its printed representation and back again.
- Convert infix notation to prefix notation.
- Use the DRAW program to gain greater understanding of the relationship between the printed representation and the pictorial representation of AutoLISP objects.

Introduction

The pictures that we have seen are a fairly close approximation of what is inside of AutoLISP. In fact, they much more accurately depict our data structures than anything we can type in. However, since we need to have a way of getting objects into and out of the machine, we must use a *printed representation* of these objects to communicate with AutoLISP.

`print` uses the printed representation of AutoLISP objects to send them to our screen. Likewise, there is a reader that takes what we type and builds AutoLISP objects internally. The I/O functions are the only functions that know about printed representation.

The pictorial representation is used throughout this book because it's both more accurate and more explanatory than the printed representation. However, to utilize it productively, we must be able to convert from the pictures to the printed representation and back again.

The following sections describe the printed representations of the AutoLISP objects that we have seen so far.

2.1 Numbers , Strings, Symbols, and Functions

Numbers, strings, symbols, and functions have fairly straightforward printed representations.

Numbers

The printed representation of a number is simply the number itself. Numbers print exactly as they have been shown thus far:

```
3
-4.56532
2.62e9
```

Real numbers must have both an integer and a fractional part. If we enter the number .7 instead of 0.7, AutoLISP issues an "invalid dotted pair" error.

Strings

Strings also print as we have seen them:

```
"Select object:"
"c:\\utils\\chgtxt.lsp"
```

The printed representation of a string is a group of characters encased in double quotes. Anytime we encounter anything in double quotes, we immediately recognize it to be a string. If it is not in double quotes, then it is not a string.

Symbols

The printed representation of a symbol is merely the name of the symbol:

```
DEFUN
SETQ
+
```

A symbol name is always printed by AutoLISP in uppercase. We can enter symbol names in upper, lower, or mixed case, and they are converted to uppercase:

Figure 2-1

Any word or token (such as +) that appears in an AutoLISP program is a symbol, unless it is encased in double quotes (in which case it is a string).

Symbol names always appear in the text of this book in lowercase, code font for readability.

Functions

Functions print in one of two ways. When we print out a function that we have written ourselves, we see the code it contains. Built-in AutoLISP functions print in the following manner:

```
<subr: #87f01ac2>
```

We gain no meaningful information from the printed representation of subrs and, therefore, usually don't intentionally print them out. A subr is always the value of a symbol, and we access the function via that symbol. Thus, we generally don't care about a subr's printed representation.

Although subrs are occasionally printed on the terminal screen, there is no way to input them. Were we to type in the printed representation of a subr, AutoLISP would not recognize it.

2.2 Conses

The non-atomic AutoLISP object types, conses and lists, have more involved printed representations. The printed representation of a cons consists of the following:

An open parenthesis: "("
The printed representation of the car of the cons
<space><dot><space>
The printed representation of the cdr of the cons
A close parenthesis: ")"

We can create a cons by running the cons function, as follows:

```
Command: (cons "Gauge" 16)
("Gauge" . 16)
```

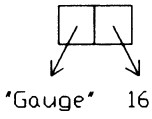


Figure 2-2

The printed representation of this cons is `("Gauge" . 16)`. Due to the dot between the car and the cdr, the printed representation of a cons is frequently called a *dotted pair.*

<dot> is the keyboard period character. It must be surrounded by spaces.

<space> is any nonprinting character, entered when we hit the <space>, <tab>, or <enter> keys. We can have as many spaces between items as we'd like; one space is equivalent to a hundred. The preceding cons could also be written

```
(   "Gauge"
.
16
)
```

We wouldn't write `("Gauge" . 16)` this way because it's unclear, but this printed representation is perfectly valid. When we begin to write programs, we will see that our ability to tab over and columnize the code facilitates its readability.

If we build a cons and forget one or more surrounding spaces, AutoLISP may not create the cons that we expect. The following examples show what gets created in a variety of cases. Ignore the quote character for now; it is explained in Section 5.1.2.4.

```
Command: (setq a '(x . y))          ;Valid way to create a cons
(X . Y)

Command: (setq a '(x .y))           ;Space missing between . and y
error: invalid dotted pair

Command: (setq lis '(1.2))          ;Builds a list whose only element is 1.2
(1.2)
```

When we enter the function `(cons 2 4)`, AutoLISP builds a new cons:

```
Command: (setq c (cons 2 4))
(2 . 4)
```

If we enter the exact same function a second time, AutoLISP creates a brand new cons but utilizes the same numbers:

```
Command: (setq d (cons 2 4))
(2 . 4)
```

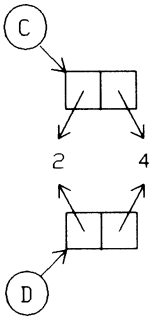


Figure 2-3

See Section 6.1.1 to learn how to differentiate between two different conses that print the same.

2.3 Lists

The printed representation of a list consists of the following:

An open parenthesis: "("
The printed representation of each element of the list separated by one or more spaces
A close parenthesis: ")"

We can create a list by running the `list` function, as follows:

```
Command: (list 1 "Wall" 2 "Ceiling" 3 "Floor")
(1 "Wall" 2 "Ceiling" 3 "Floor")
```

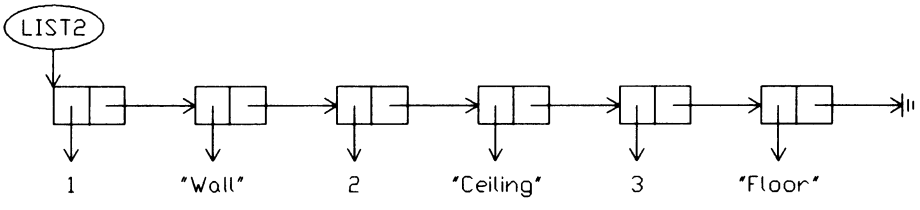


Figure 2-4

The printed representation of this list is `(1 "Wall" 2 "Ceiling" 3 "Floor")`.

Since lists are objects that we create frequently, it is extremely important that we gain an understanding of their structure and become facile in their manipulation. Before going further in the study of AutoLISP we *must* learn the following:

- How lists are created and their appearance in both pictorial and printed notation
- How to convert between these two representations
- How to use such functions as `car`, `cdr`, `cadr`, and `cddr` to move through lists and extract information from them

Using the pictures along with the printed representations may at first seem unnecessarily time consuming. However, the pictures are not only more accurate than the printed representation, they also illustrate what is happening in AutoLISP far better than printed data and code possibly can. I promise that time spent learning how to work with these pictures will pay off manyfold down the road.

The following examples demonstrate how to convert from the pictorial to the printed representation of lists.

Example 1

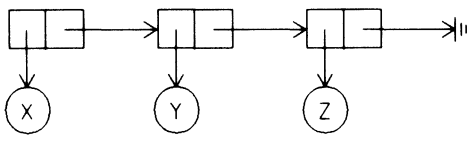


Figure 2-5

This is a simple list of three elements, `x`, `y`, and `z`. From the definition of a list's printed representation we determine that the list prints as `(x y z)`.

Example 2

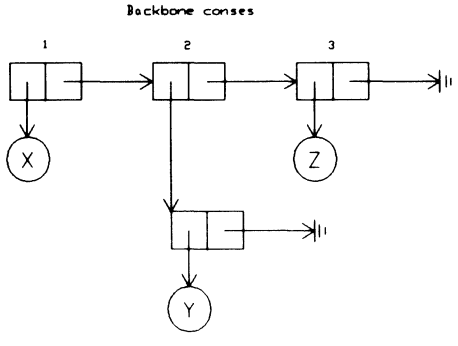


Figure 2-6

To ascertain the printed representation of a list, we must first determine how many elements it has. The easiest way to accomplish this is to count the number of conses in the backbone. The backbone of this list has three conses, which means that the list has three elements.

The first element of the list is `x` and the third element is `z`. We can write what we've determined so far as

```
(x something z)
```

where *something* represents an AutoLISP object that is unknown at this point. We can draw what we have so far as follows:

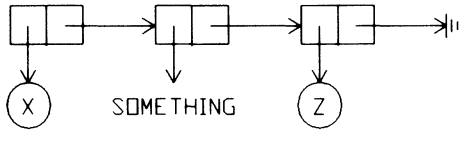


Figure 2-7

We know that the second element is *something;* we just don't know yet what it is. Let's examine this element.

The second element is a list. There's one cons in its backbone, so its length is one. In fact, it is a list whose only element is `y`. Its printed representation is

```
(y)
```

It looks like this:

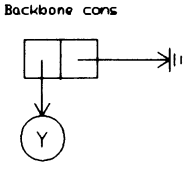


Figure 2-8

To complete the printed representation of the entire list we simply substitute `(y)` in place of *something,* giving us

```
(x (y) z)
```

The printed representation of this list has nested parentheses because one of the list's elements is itself a list. Since any AutoLISP object can be a member of a list, this list is quite acceptable.

Note: There are two tiers of conses in Figure 2-6 and there are two levels of parentheses in its printed representation. The number of cons tiers will *always* equal the number of levels of parenthesis nesting.

Note: For the remaining examples, try converting from pictorial to printed representation before reading the explanation. Be certain you understand an explanation before going on to the next example.

Example 3

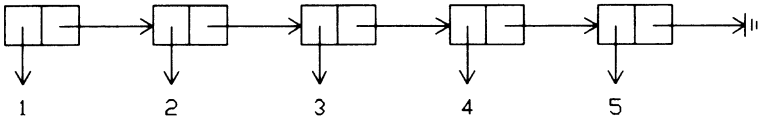


Figure 2-9

This is a simple list of five elements, 1, 2, 3, 4, and 5. Its printed representation is `(1 2 3 4 5)`.

Example 4

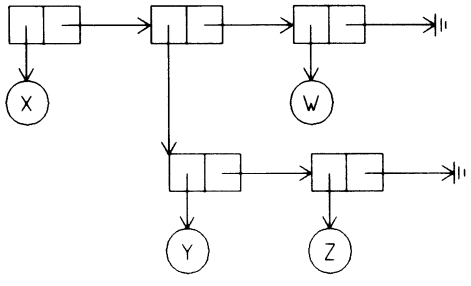


Figure 2-10

This is a list of three elements, because there are three conses in the backbone. The first element of this list is the symbol `x` and the third element is `w`. We can write what we have so far as

```
(x something w)
```

It looks like this:

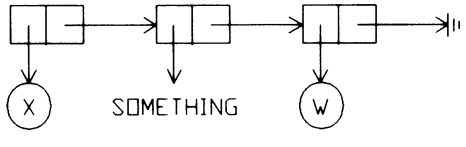


Figure 2-11

The second element is a list:

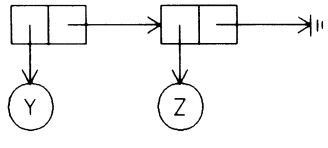


Figure 2-12

There are two conses in its backbone so its length is two. Its elements are `y` and `z`, and its printed representation is

```
(y z)
```

To complete the printed representation of the entire list we simply substitute `(y z)` in place of *something*, giving us

```
(x (y z) w)
```

Note: Our initial impulse is to look at the picture of this list and say that it has four elements, `x`, `y`, `z`, and `w`. But including `y` and `z` would require us to trace through arrows, something that we never do. When trying to determine a list's length, counting atoms is often misleading. This is actually a three-element list whose second element is the sublist `(y z)`. If we count the backbone conses we always learn the correct length of a list.

Example 5

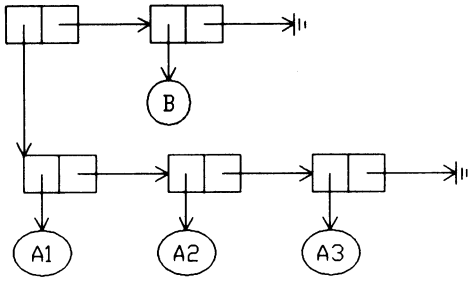


Figure 2-13

There are two conses in the backbone, which means that this list has two elements. The first element is *something* and the second element is `b`. We can write what we have so far as

```
(something b)
```

It looks like this:

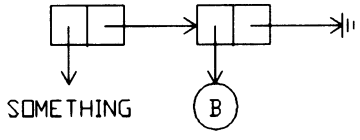


Figure 2-14

The first element is itself a list of three elements, `a1`, `a2`, and `a3`.

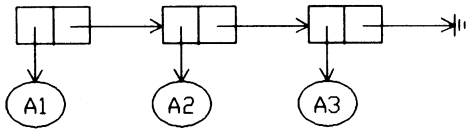


Figure 2-15

Its printed representation is

```
(a1 a2 a3)
```

Once again, we complete the printed representation of the entire list by substituting `(a1 a2 a3)` in place of *something*, giving us

```
((a1 a2 a3) b)
```

Notes

- We always translate lists from picture notation to printed representation by going from left to right; the printed representation of this list is *not* `(b (a1 a2 a3))`. For that to happen `b` would have to be the car of the first backbone cons, which is clearly not the case in this example.

- This is the first printed representation we have seen that starts with two consecutive left parentheses. This phenomenon occurs whenever we have a list whose initial element is itself a list.

Example 6

This is the same picture as the last one, but this time the symbol `b` has a value of 5. How do you think that will affect the printed representation? This is something we've not yet explicitly discussed, so think about it a little, then write down your best guess.

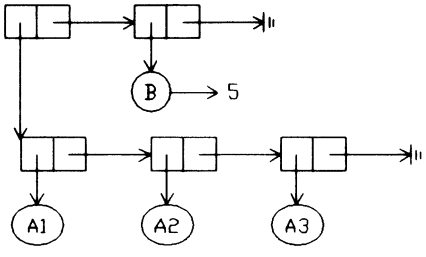


Figure 2-16

Most people respond with one of these two solutions: Either they state that the printed representation stays the same:

```
((a1 a2 a3) b)
```

Or they say that the symbol `b` is replaced by its value, 5:

```
((a1 a2 a3) 5)
```

If you said that the printed representation doesn't change, you were right! Here's a rule regarding the printed representation of symbols:

> The value of a symbol never appears in the printed representation of that symbol.

How might we have known this? Well, what's the value of the symbol `a1`? Since `a1`'s value is not shown, we know that it is `nil`. `a2` and `a3` have `nil` as their values as well. If printed representation showed symbols' values instead of the symbols themselves, then this list would print as follows:

```
((nil nil nil) 5)
```

Since any variable can have `nil` as its value, AutoLISP would not be able to construct the original list from this printed representation. One reason that pictures are better than printed representation is that they display *all* of the relationships among AutoLISP objects; printed representation does not tell the entire story.

Example 7

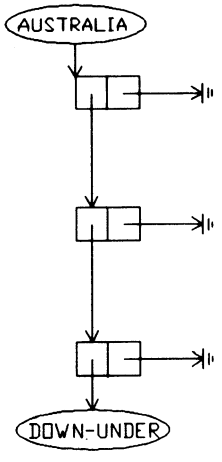


Figure 2-17

This example is more easily understood bottom-up. The list at the bottom has only one element, `down-under`, so its printed representation is

```
(down-under)
```

The cons in the middle is a one-element list whose only element is the list we've just examined. Its printed representation is a set of parentheses containing the printed representation of this list:

```
((down-under))
```

The top-level list is a list containing only the middle-level list. Thus, its printed representation is

```
(((down-under)))
```

We've seen that lists are formed by linking conses together via their cdrs. This example shows the effects of linking conses by their cars. It's not a pretty sight, but it's certainly legal. We'll frequently nest lists several levels deep, but they'll have more than one element and won't appear so strange.

Note: The symbol `australia` does not appear in the printed representation because it is not part of this list. To include `australia` would require us to backtrace an arrow (i.e., go against the way it points), something we cannot do.

Example 8

This last example's a toughie! If you can do this one, you can do any of them.

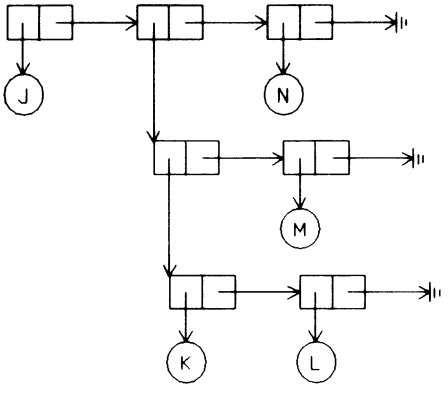


Figure 2-18

We know this list has three elements because there are three conses in the backbone. Furthermore, since the conses are nested three levels deep, we must show three levels of parenthesis nesting when we write the printed representation.

This problem may be solved using a top-down or bottom-up approach. Both methods are shown here; choose the one that makes the most sense to you.

Top-Down Approach

The top-level list has three elements, `j`, *something,* and `n`:

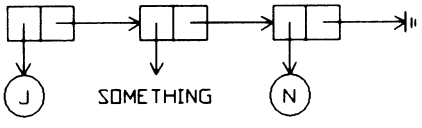


Figure 2-19

```
(j something n)
```

Now we must determine how *something* prints. It is a list of two elements, *something* and `m`.

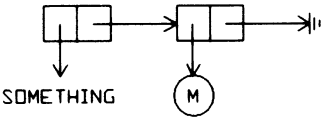


Figure 2-20

```
(something m)
```

Since this is the printed representation of the second element of the top-level list, it can be inserted in that list's printed representation:

```
(j (something m) n)
```

Notice that what we have so far looks remarkably similar to the solution to Example 4. This makes sense because the first two cons tiers have the same pattern as Figure 2-10.

The last *something* is a two-element list of the symbols `k` and `l`:

```
(k l)
```

When we substitute this printed representation for the last *something,* we get the printed representation of the entire structure:

```
(j ((k l) m) n)
```

Notes

- Each substitution is made in its entirety; parentheses are not removed. This structure just happens to encompass a list whose first element is another

list, which causes the printed representation to contain two consecutive open parentheses.

- Although there are many conses in the drawing, this is still a list of three elements. To determine a list's length, count backbone conses, not atoms.

Bottom-Up Approach

Look at the very bottom of Figure 2-18. It is a simple list of two elements, `k` and `l`. Therefore, its printed representation is

```
(k l)
```

The list above `(k l)` is also a two-element list. Its first element is the list `(k l)`, and its second element is `m`. Thus, its printed representation is

```
((k l) m)
```

Finally, the top-level list has three elements: `j`, the list we just wrote, and `n`. Therefore, its printed representation is

```
(j ((k l) m) n)
```

Notes

- It doesn't matter whether we use a top-down or bottom-up approach, as long as we derive the correct printed representation. Choose the method that best suits your style.
- When first learning this translation process, it appears that there are an infinite number of possibilities for the printed representation of lists. Actually, very few choices are possible:
  - We can have a simple list, such as `(1 2 3)`.
  - A list can have one or more elements that are lists, such as `(1 (2 3 4) 5)` or `(1 (2 3) (4 5) 6)`. If an element is a cons, then the printed representation will contain a dotted pair: `(1 (2 . 3) 4)`.
  - The first element of a list can be a list, giving us two open parentheses in a row: `((1 2) 3)`. The completely vertical list we examined in Example 7 is a highly unusual occurrence.
- Most lists that we write will appear like one of these. Furthermore, it doesn't matter how many levels of conses we have; these patterns repeat. For ex-

ample, the exercise we just did initially appears to be intricate but is actually quite similar to the lists in Examples 4 and 5. If you break lists into their components, you'll quickly gain familiarity with these patterns, and the process of translating from picture to printed representation will become second nature.

♦ ♦ ♦

We can write the printed representation of the list in Example 8 as follows:

```
(j((k l)m)n)
```

AutoLISP recognizes parentheses as delimiters, so spaces are not needed between atoms and parentheses. However, improper formatting makes data and code much harder to read. The convention for printing lists is to place a space between an atom and an open parenthesis:

```
j (
```

and between a close parenthesis and an atom:

```
) m
```

Don't put a space between an open parenthesis and an atom:

```
(j
```

or between an atom and a close parenthesis:

```
n)
```

If list elements are atoms, however, we must separate them with spaces. The list

```
(kl)
```

is a one-element list containing the symbol `kl`; it's *not* a two-element list of `k` and `l`.

2.4 Converting Printed Representation to Picture Notation

It is not sufficient to be able to convert from pictorial to printed representation; we must also be able to convert from a printed representation to its pic-

ture. As we shall see, the pictorial representation can aid us tremendously in understanding and accessing data structures, but only if we know how to take a printed representation and draw the picture.

The pictorial representations were introduced in Chapter 1 and, except for lists, should be pretty straightforward. Dotted pairs, you'll recall, are the printed representations of conses. Thus,

```
(a . b)
```

is drawn like this:

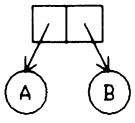


Figure 2-21

The rules for converting from the printed representation to the picture notation of lists aren't easily expressed. This technique is best learned by examples and practice.

Before examining nested lists, we *must* know that the picture notation for what we call a "simple list," such as

```
(x y z)
```

is

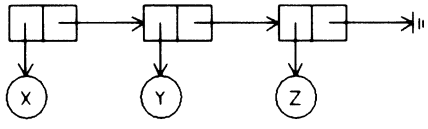


Figure 2-22

All other list pictures are just expansions and repetitions of this pattern. A few examples will clarify the technique. In each example, once we have finished drawing the picture, convert it back to its printed representation according to the rules we learned in the last section. This will verify that the two representations are indeed the same list.

Example 1

Let's draw the picture for this printed representation:

```
(left (middle) right)
```

Because the entire printed representation is in parentheses, we know that this object is a list. In this list the first thing we encounter is the symbol `left`. Thus, `left` is the first element of the list.

The second thing we encounter is an open parenthesis, which tells us that the second element of the list is itself a list. Everything between that open parenthesis and the *matching* close parenthesis constitute the inner list, although at this point we can't see the inner list's elements. We use the term *something* to specify that we don't yet know what's within the parentheses:

```
(left (something) right)
```

The third thing we encounter is the symbol `right`, making it the third element of the list. Following `right` is the close parenthesis that tells us we're at the end of the list. The picture we have so far is

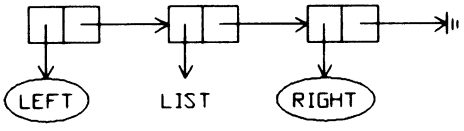


Figure 2-23

The inner list is a simple one-element list consisting of the symbol `middle`. Replacing *something* with its picture, we get the final pictorial representation of this list:

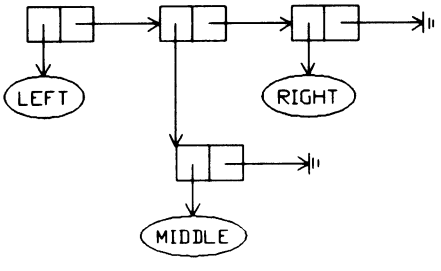


Figure 2-24

Example 2

```
(a (b (c)))
```

This list is more intricate than the last, but the steps we take are the same. The first thing we see after the open parenthesis is the symbol a. That's the first element of the list.

The second thing we encounter is an open parenthesis, which tells us that the second element of the list is itself a list. Everything between that open parenthesis and the *matching* close parenthesis constitute the inner list:

```
(a (something))
```

We now reach the close parenthesis for the top-level list, which means that we have a two-element list that looks like this:

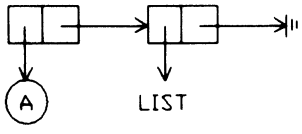


Figure 2-25

We must next create the picture for the second-level list. In this list the symbol b is the first thing we encounter, so it's the first element. Following the b is an open parenthesis, which tells us that the second element is another list. Everything between the open parenthesis and the matching close parenthesis constitute the inner list:

```
(b (something))
```

We now reach the close parenthesis for the second-level list, which means that it also has two elements:

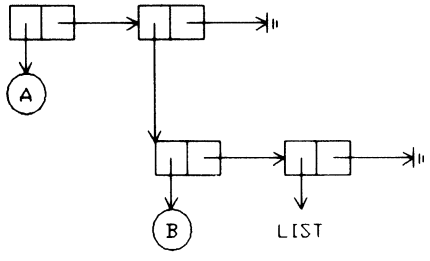


Figure 2-26

The innermost list is a simple one-element list containing the symbol `C`. Replacing *something* with its picture, we get the final pictorial representation of this list:

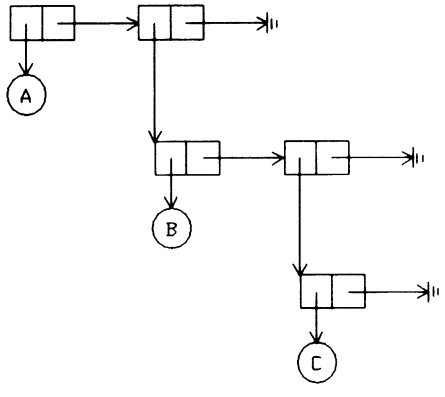


Figure 2-27

Notice that the printed representation of this list has parentheses nested three levels, and the picture of the list has three cons tiers. The number of cons tiers will *always* equal the number of levels of parenthesis nesting.

Important Notes

- When we convert from printed to pictorial representation, it is important to recognize that everything between an open parenthesis and the *matching* close parenthesis constitute an inner list. But what do we mean by *matching parentheses?*

  A pair of parentheses match if there is an *equal* number of open and close parentheses between them. The parentheses are bracketed correctly in the list on the left, incorrectly in the one on the right:

  The salient feature of the correctly bracketed list is that the lines don't cross. The easiest way to bracket a list is to start with the innermost parentheses and work outward.

- When AutoLISP looks at this list it doesn't see all of the symbols contained therein. It only sees this:

```
(a b (something) f)
```

It realizes that `a` and `b` are the first two elements and that `f` is the fourth. But when it encounters the open parenthesis following `b` all it knows is that the third element is a list. It doesn't look inside the list because it would have to trace through arrows to do so. The third element is just one *list*, regardless of how many elements or how many levels of cons nesting it might have. We'll draw this list in Example 5.

Note: The remaining examples give you the opportunity to practice converting from printed to pictorial representation before the explanation is given. Draw the picture for each of these lists, and be certain you understand an explanation before going on to the next example.

Example 3

For this example, after you've drawn the picture determine the car, cdr, and cadr of the list. Also, what sequence of cars and cdrs are needed to access the symbol `g`?

```
(d e (f . g) h)
```

This list contains a dotted pair, which is the printed representation of a cons. We can simplify the list as follows:

```
(d e cons h)
```

This is a four-element list whose third element is the cons `(f . g)`. Here's the pictorial representation of this list:

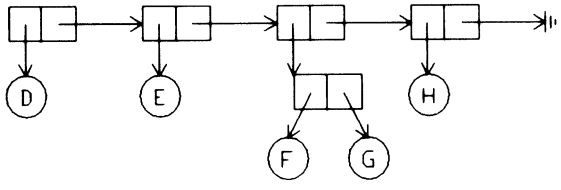


Figure 2-28

- The car of this list is `d`. The car is always the first element of a list.

- The cdr of this list is `(e (f . g) h)`. The cdr of a list is the second cons in the backbone; this is the printed representation of that cons.

  Here's an easy way to identify the cdr: the cdr is always the exact same list we began with, if we eliminate the car. If we remove `d` from `(d e (f . g) h)`, we end up with `(e (f . g) h)`. Once we know the car, we can always determine the cdr in this fashion.

- The cadr of this list is `e`. *cadr* is short for *car of the cdr.* Having learned that the cdr is `(e (f . g) h)`, we simply take the car of this list. Even more simply, the cadr is always the second element of a list.

- The symbol `g` is the cdr of the car of the cdr of the cdr (or cdaddr) of this list. We arrive at this result by starting with `g` and working backward to the first cons:

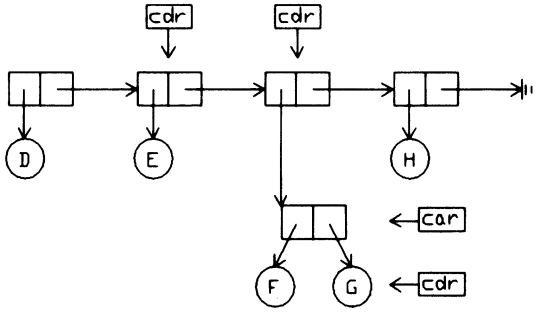


Figure 2-29

Example 4

```
((1 2) (3 4))
```

This list begins with two open parentheses in a row, which tells us that the first element of the list is itself a list. Everything between the second open parenthesis and its matching close parenthesis constitutes this inner list. It is the list `(1 2)`:

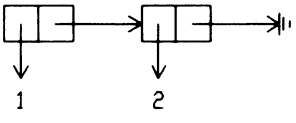


Figure 2-30

Following this list we encounter another left parenthesis, which means that the second element of the list is also a list. It's the list `(3 4)`:

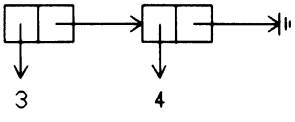


Figure 2-31

Since we now encounter the close parenthesis that ends the top-level list, we're done. What we have is a two-element list, and both elements are themselves lists:

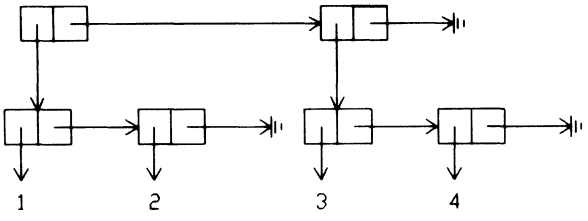


Figure 2-32

Example 5

```
(a b (c ((d) e)) f)
```

As we discussed in the note following Example 2, this is a list of four elements:

```
(a b (something) f)
```

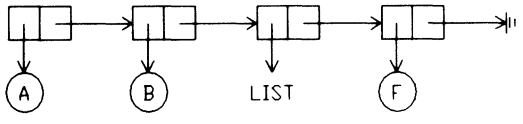


Figure 2-33

Now let's examine (*something*), which is the inner list, `(c ((d) e))`.

The first element of this list is `c`. Next comes an open parenthesis, indicating that the second element is a list. The *matching* close parenthesis immediately follows the symbol `e`. The final close parenthesis marks the end of this two-element list, giving us the following intermediate expression:

```
(c (something))
```

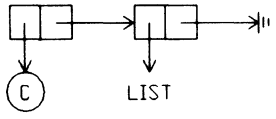


Figure 2-34

This intermediate expression can be added to our original one, giving us

```
(a b (c (something)) f)
```

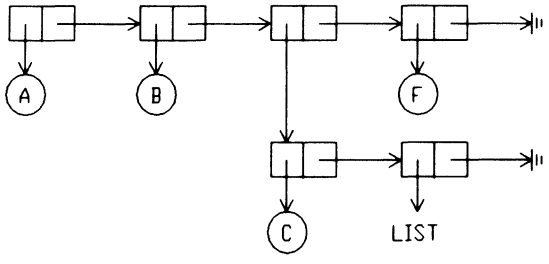


Figure 2-35

We now have another list to examine, `((d) e)`. The two consecutive open parentheses tell us that the first element of the list is yet another list, whose only element is `d`. The second element of the list is `e`. The list is drawn in the following way:

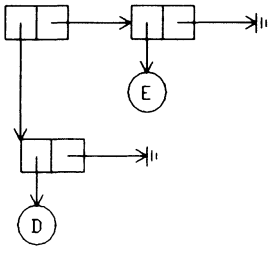


Figure 2-36

Now we can put all the parts together and get the complete drawing for this list:

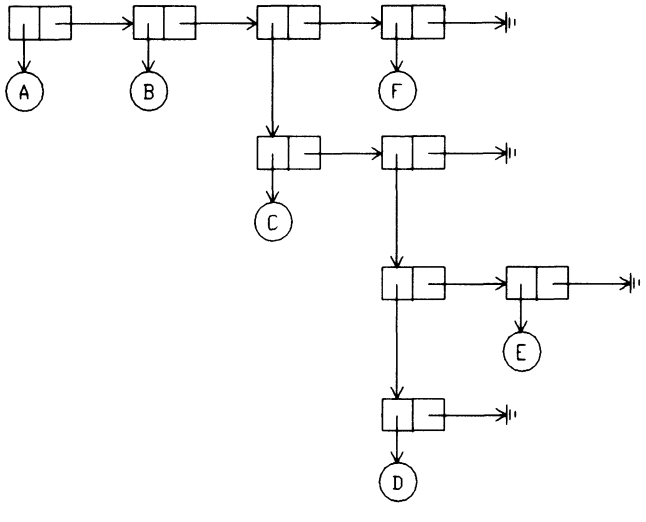


Figure 2-37

To verify this, use our rules to translate this picture back to its printed representation. Notice again that there are four levels of parentheses and a matching four tiers of conses.

♦ ♦ ♦

Here's a point that was made earlier but is worth repeating, because it frequently causes confusion for new AutoLISP users. Look again at a list we saw earlier, `(x y z)`:

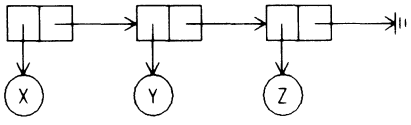


Figure 2-38

The first cons is the list `(x y z)`, the second cons is the list `(y z)`, and the third cons is the list `(z)`. For convenience we often state that the list `(x y z)` is the entire structure, whereas it is really the *first cons*. Remember, a list is *a cons* that we can cdr to `nil`.

Every cons must have both a car and a cdr, and the cons cannot be divorced from either of these. Therefore, when we access the cons whose car is `x`, the other two conses come along for the ride. In other words, a function cannot return the cons whose car is `x` without returning the next two conses as well. For ease of discussion we assume that the list `(x y z)` is the entire structure, but to AutoLISP it is the first cons only. We must remember this when we begin to program so that we can correctly anticipate our functions' results.

2.5 Background Topic: Rules for Simplifying Lists

You might have noticed a curious fact regarding the printed representation of lists. A list is merely a collection of conses that are linked by their cdrs, ending in `nil`. However, the printed representation of lists and conses are radically different. Conses split their components with a dot, but lists don't. The last cons in a list has `nil` as its cdr, but the `nil` does not appear in the printed representation. Thus arises the question, what happens to the dots and the `nil` in the printed representation of lists?

The simple answer is, it doesn't matter. We can program in AutoLISP perfectly well without ever learning what happens to the dots and the `nil` in the printed representation of lists. In fact, if you're just beginning to learn AutoLISP, I sug-

gest that you skip this Background Topic for now and come back to it at a later date. Although it's not difficult, there's much more important material to master at this time.

If your curiosity has been piqued, however, here's the real answer: There are rules that simplify the printed representation of list structure. We use the simplified version without thinking about it, but sometimes list structure isn't printed as we might expect. Then it's useful to understand just how AutoLISP actually does derive the printed representation of lists.

Consider the following list:

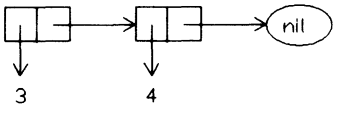


Figure 2-39

Using the rules for printing conses, the printed representation of the cons on the right is

```
(4 . nil)
```

The printed representation of the cons on the left is

```
(3 . (4 . nil))
```

Notice that a cons comprises both its car and its cdr. Thus, even though the cons on the left is being printed, its printed representation includes the cons on the right.

Imagine how tiresome this notation would be to print if the list had even 10 members! Fortunately, the following two *reduction rules* simplify the printing of lists:

1. Remove all occurrences of ". `nil`".

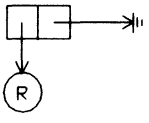


Figure 2-40

According to the rules for printing conses, the printed representation of this cons is `(r . nil)`. When the first reduction rule is applied, it simplifies it to `(r)`. Similarly, in Figure 2-39, `(3 . (4 . nil))` reduces to `(3 . (4))`.

2. Remove all occurrences of ". (" and the matching ")".

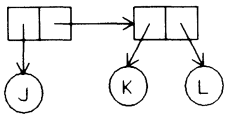


Figure 2-41

According to the rules for printing conses, the printed representation of this structure is `(j. (k . l))`. Application of the second reduction rule simplifies it to `(j k . l)`. The list in Figure 2-39, which was semireduced to `(3 . (4))` by the first reduction rule, reduces further to `(3 4)`.

Notes

- The two rules can be applied in either order.
- The structure shown in Figure 2-39, `(3 4)`, is a list; it reduces nicely to the printed representation of the elements encased in parentheses.
- Sublists reduce similarly.

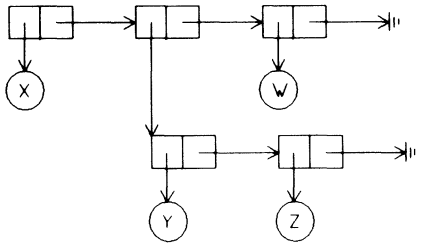


Figure 2-42

The expanded printed representation of this list is `(x . ((y . (z . nil)) . (w . nil)))`. By applying the reduction rules we get the more familiar and accessible `(x (y z) w)`. It's a list of three elements, one of which is the sublist, `(y z)`.

- Why doesn't `(j k . l)` in Figure 2-41 lose all of its dots? When we examine the picture we see that it is *not a list.* The last cons has a cdr of `l`, not

`nil`. The printed representation of this cons is `(k . 1)`; the dot doesn't disappear.

2.6 Prefix Notation

When writing AutoLISP programs, we accomplish tasks by applying operators to operands.

An *operator* is a symbol that specifies an action to be performed. In other words, it is a symbol whose value is a function. Arithmetic symbols such as +, –, * (multiplication), / (division), > (greater than), and < (less than) are examples of operators. So are the names of the basic functions we've seen, such as `car`, `cdr`, `cadr`, `cons`, `setq`, and `list`.

An *operand* is the data on which operators perform their actions. For example, in the expression 2 + 3, the symbol + is the operator and the numbers 2 and 3 are the operands.

In everyday life (as well as in most programming languages), we specify mathematical operations by putting an operator between the operands it's operating on. This is called *infix notation.* Familiar examples include:

```
x = 5
x = x + 1
average = (n1 + n2 + n3) / 3
```

In AutoLISP, on the other hand, we write expressions using a method called *prefix notation.* AutoLISP expressions written in prefix notation conform to the following rules:

- In every expression or subexpression, the operator is *always* placed first and all operands follow.
- The entire expression (and each subexpression) is surrounded by parentheses.
- `setq` replaces the = assignment operator used in infix notation.

The preceding infix notation examples are shown below, this time expressed in prefix notation:

```
(setq x 5)
(setq x (+ x 1))
(setq average (/ (+ num1 num2 num3) 3))
```

Background Note: Prefix notation was invented a number of years ago by the Polish logician Jan Lukasiewicz. Scientists wanted to name this methodology after its inventor but, because they found his name difficult to pronounce, began calling it *polish notation!* This notation is used by compiler writers when they parse expressions and is familiar to many of us as the basis of Hewlett-Packard calculators. The calculators differ from AutoLISP in that they require us to supply the operator *after* the operands; this is known as *reverse polish notation.* The term *polish notation* has been replaced by the more politically correct *prefix notation,* although it's still an eye-catcher when your résumé says you speak *reverse polish!*

When we want to run a function in AutoLISP, we can type it at the Command: prompt in a style similar to these examples.

It's important to note that when we convert from infix to prefix notation, we always list *first* the operation that we want performed *last.* To say the same thing differently, AutoLISP always executes the code that's in the innermost parentheses first.

In the last expression above, the three numbers are added together first, then divided by 3. Thereafter the quotient is assigned to the variable `average`. Note that although the operator is moved to the front of each subexpression, the order of operands does not change. The following expression reverses the operands' order and yields a very different result:

```
(setq average (/ 3 (+ num1 num2 num3)))
```

The following examples walk us through the process of converting from infix to prefix notation:

Example 1

```
radians = pi * angle/180.0
```

The last action performed by this expression is to assign a result to the variable `radians`. Therefore, the expression *starts* with

```
(setq radians
```

This completes the left side of the assignment. Now let's look at the right side. Here, the last thing to be done is the division; some *sub-expression* must be divided by 180.0. What we have so far is

```
(setq radians (/ sub-expression 180.0))
```

Finally, the sub-expression, `pi * angle`, is converted to prefix notation:

```
(* pi angle)
```

Put this all together and we have the prefix notation for the entire expression:

```
(setq radians (/ (* pi angle) 180.0))
```

It is interesting to note that the expression

```
(setq radians (* pi (/ angle 180.0)))
```

returns the same result because these arithmetic operators are commutative (can be written in any order). This is usually not the case with AutoLISP expressions; we generally must be very careful about the order in which we specify operands. On the other hand, we'll find that there are frequently two or more ways to accomplish the same task in AutoLISP. Some solutions are more elegant than others, but there is rarely a "right" way and a "wrong" way if two solutions give the correct result.

Example 2

```
result = ((a * 4) - (b * 6)) / c
```

The last thing to be done is to assign a solution to the variable `result`. Therefore, the expression begins with

```
(setq result
```

On the right side of the assignment the last thing that must be done is the division; some *sub-expression* must be divided by the symbol `c`. We can write what we have so far as follows:

```
(setq result (/ sub-expression c))
```

In the *sub-expression,* the last task to be performed is subtraction. We can write this *sub-expression* as follows:

```
(- term1 term2)
```

term1 is the multiplication of `a` by 4; it's written

```
(* a 4)
```

term2 is the multiplication of b by 6; it's written

```
(* b 6)
```

Thus, the subtraction *sub-expression* is written

```
(- (* a 4) (* b 6))
```

Finally, we can substitute this *sub-expression* in the original expression, giving us the complete prefix notation:

```
(setq result (/ (- (* a 4) (* b 6)) c))
```

Notes

- When we're at this point in the conversion process

  ```
  (setq result
  ```

  we look at the right side of the equation and ask ourselves, "What's the last thing we need to do here?" The answer is "divide," but there is a temptation to say "divide by c." This is a potentially dangerous way to think, because we could end up writing the following expression:

  ```
  (setq result (/ c (- (* a 4) (* b 6))))
  ```

  In other words, we might divide c by the *sub-expression* rather than the other way around, which gives us the reciprocal of the correct result. This doesn't happen when we apply the following rule:

  > In prefix notation, the operators move to the front, but the order of operands remains the same.

  In the original infix expression, the order of operands is result, a, 4, b, 6, and c. They remain in that order when we convert to prefix notation.

- There is *always* an open parenthesis before an operator, and there is *never* an open parenthesis before an operand. This is an immutable fact in AutoLISP; learning it now will save you a lot of pain in the future.

In the next two examples, try converting the expression to prefix notation before reading the explanation.

Example 3

```
(x * 2) + (y * 3) + (z * 4)
```

The last task to be performed by this expression is addition. One benefit of prefix notation is that we can add as many expressions as we want, like so:

```
(+ sub-expression sub-expression . . . sub-expression)
```

The three sub-expressions are

```
(* x 2)
(* y 3)
(* z 4)
```

We put these together to get our result:

```
(+ (* x 2) (* y 3) (* z 4))
```

It wouldn't be wrong to do three separate additions; this way is just simpler.

Example 4

```
s = ((v1 + v2) / 2) * time
```

In this expression the addition is performed first, followed by the division, the multiplication, and the assignment to `s`. Therefore, we write the prefix notation in reverse order, beginning with

```
(setq s
```

Next, we put down what we can for the multiplication:

```
(setq s (* sub-expression time))
```

Now we must look at the *sub-expression.* It consists of the following division:

```
(/ term 2)
```

The *term* is an addition of two variables

```
(+ v1 v2)
```

that can be substituted into the *sub-expression,*

```
(/ (+ v1 v2) 2)
```

which can now be placed back into the original expression, giving us the full prefix notation:

```
(setq s (* (/ (+ v1 v2) 2) time))
```

2.7 The DRAW Program

The floppy disk enclosed with this book includes a file called `draw.lsp` that contains an on-line tutorial named DRAW. When we supply the printed representation of an AutoLISP object to the DRAW program, it draws the pictorial representation of that object on the screen.

The DRAW command is an extremely valuable aid for learning how printed representation corresponds to picture notation. Copy this file into your current directory, then load it into AutoLISP as follows:

```
Command: (load "draw")
```

Run the DRAW command by entering

```
Command: draw
```

When the program prompts, enter the list `(a b c)` as follows:

```
Enter object to draw: (a b c)
```

The following should appear on your screen:

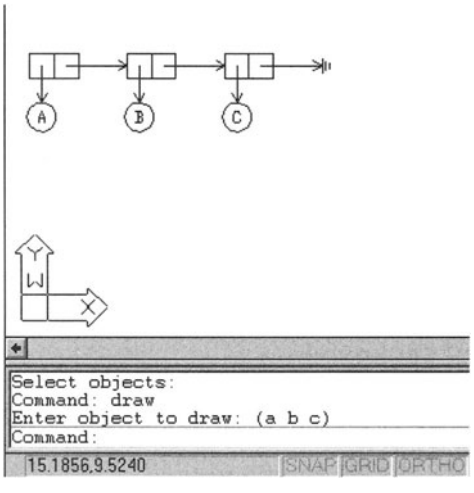


Figure 2-43

If you want to run the command a second time, simply press the <enter> key and, like any AutoCAD command, DRAW will be executed again.

Notes

- The DRAW program can draw any AutoLISP datatype. It is possible to create complex lists that visually overlap and are hard to read, or extend beyond the screen. If a list's members are symbols with very long names, the names may overlap as well. For the sake of code brevity, I haven't addressed every possible input. However, most lists will print quite nicely. If a portion of the list isn't visible, simply ZOOM EXTENTS.
- Run DRAW on the standard AutoCAD screen. The program erases anything that is already on the screen.
- The DRAW program creates the file `temp.dat`; it can be deleted.

Let's try a few examples. Initialize the AutoLISP world by setting the values of `a`, `b`, and `c` to 1, 2, and 3, respectively, as follows:

```
Command: (setq a 1)
1

Command: (setq b 2)
2

Command: (setq c 3)
3
```

Next, run the DRAW program and enter the following lists:

```
Command: draw
Enter object to draw: (a (b . c) d)
```

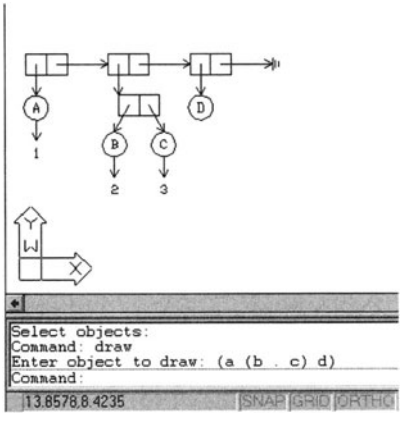


Figure 2-44

```
Command: draw
Enter object to draw: (a (b (c) d) "efg")
```

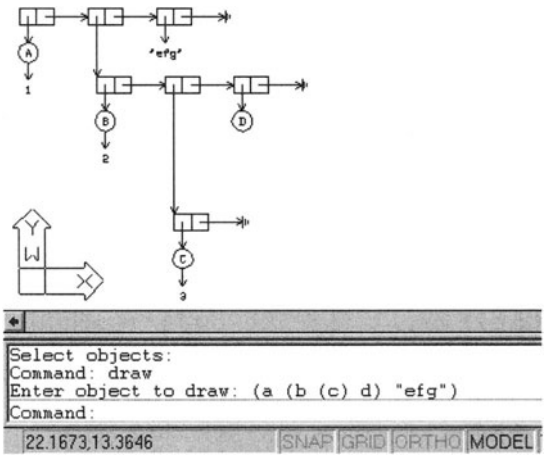


Figure 2-45

2.8 Important Points to Remember

- Real numbers require digits on both sides of the decimal point. If we supply .3 instead of 0.3 to AutoLISP, we get an "invalid dotted pair" error.

- AutoLISP *always* prints symbol names in uppercase (whereas they are lowercase code font in this book). Strings are *always* surrounded by double quotes. *Everything* in a program listing that looks like an English-language word is a symbol unless it is encased in double quotes.

- The value of a symbol *never* appears in the printed representation of that symbol.

- The car of a list is always the first element, the cadr is always the second element, and the caddr is always the third element. These correspond to the X, Y, and Z coordinates of a point.

- The cdr of a list is always the second cons in the backbone of the list. The cdr is the exact list we started out with minus the car.

- Use one or more spaces to separate elements of a list or the car, dot, and cdr of a cons. `(1(2)3)` is valid but hard to read. `(123)` is a one-element list containing the number 123.

- Prefix notation requires that an operator precede each of its operands. The expression `(+ 3 4)` adds the numbers 3 and 4. Similarly, `(list 5 7)` creates a list containing the elements 5 and 7. On the other hand, `(3 + 4)` and `(5 list 7)` have no meaning in AutoLISP and will produce errors if we type them in.

  This rule holds true for nested lists as well. In the expression `(+ 3 (* 4 5) 6)`, the operator is + and the operands are 3, `(* 4 5)`, and 6. Nested expressions, no matter how deep the nesting, must also be specified using prefix notation. It is incorrect to enter `(+ 3 (4 * 5) 6)`.

- When converting from infix to prefix notation, the operators are moved to the front, but the order of operands remains the same.

- An operator, such as `list`, `+`, or `setq`, is a symbol. The value of that symbol is a function.

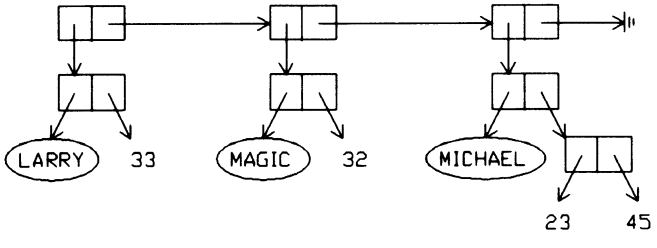
- An operator always follows an open parenthesis; an operand never does.

- Don't use the DRAW program to solve the following labs! You'll derive more from the exercises if you work them out on your own, then use the DRAW program to check your answers.

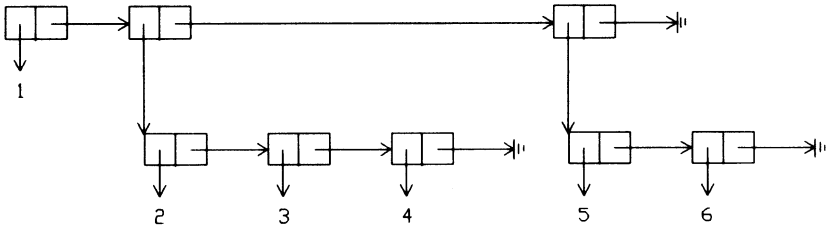
2.9 Labs

1. Convert the following pictures into their printed representations. Then check your work by converting the printed representations back to pictures with the DRAW program.

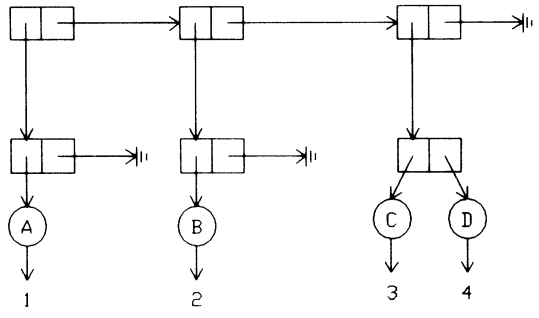
a)



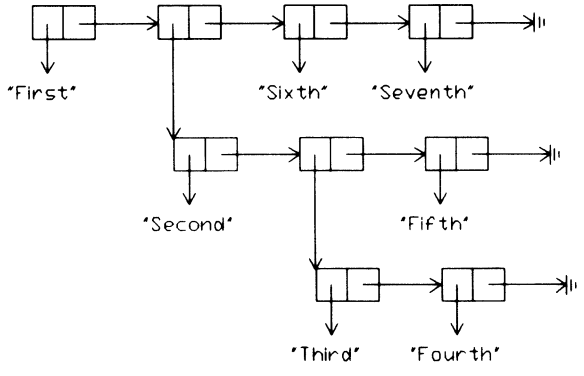
b)



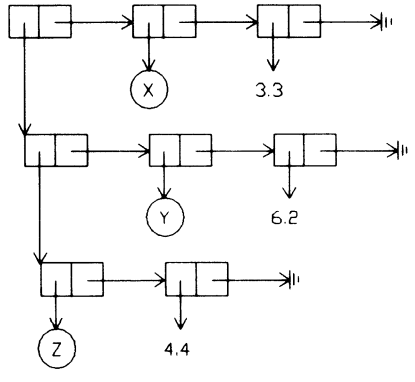
c)



d)



e)



2. Convert the following printed representations into their pictures. You can use the DRAW program to check your work (or just convert back to printed representation).

   a) `(1 2.0 0.3 4 5 6.6)`

b) `(a (b c) d)`

c) `(diana (michael . janet))`

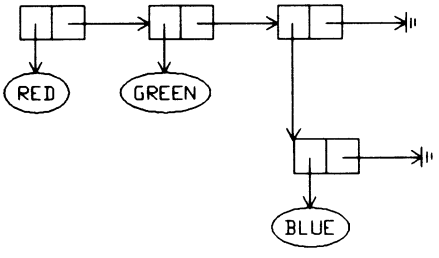
d) `((q r) (e f (g) h))`

e) `((8 (a 3 (2 (d . e) 6) m)))`

3. For each of the following lists, determine the car, the cdr, and the cadr. What sequence of cars and cdrs will return `blue` in each of these lists? So that you understand what is expected, a sample exercise is shown.

   Note: This exercise is made infinitely easier by drawing the pictures! Try doing this on your own, without the DRAW program.

   `(red green (blue))`



   `car = red`
   `cdr = (green (blue))`
   `cadr = green`
   `blue` → car of the car of the cdr of the cdr = caaddr

   Note: It is easy to determine that `blue` is the caaddr of the list by working backward from `blue` to the first cons in the preceding picture. It is substantially harder to determine this from the printed representation of the list without the picture. As examples become more complex the advantage of using pictures will become even more apparent.

   a) `(red yellow green cyan blue magenta white)`
      `car =`
      `cdr =`
      `cadr =`
      `blue` →

b) `(("Sad" "Gloomy" "Blue") "Joyful" "Happy")`
   `car =`
   `cdr =`
   `cadr =`
   `blue →`

c) `((navy . deacon) mister (stella baby . blue))`
   `car =`
   `cdr =`
   `cadr =`
   `blue →`

d) `(indigo (cerulean (azure blue)))`
   `car =`
   `cdr =`
   `cadr =`
   `blue →`

e) `(((blue . moon)) (laws movies) "Suede shoes")`
   `car =`
   `cdr =`
   `cadr =`
   `blue →`

4. Enter each of the following AutoLISP functions at the Command: prompt and see what gets returned. Write down what you believe will be returned before you enter each one. Don't worry if you can't, because it utilizes some concepts that are presented in subsequent chapters. Hint: Draw the picture!

```
Command: (setq list1 (list 1 2 3 4))

Command: (car list1)

Command: (cadr list1)

Command: (caddr list1)

Command: (cdr list1)

Command: (cddr list1)

Command: (cdddr list1)
```

   In this example, `car`s return numbers and `cdr`s return lists of numbers. Why does this happen?

5. Draw what is created when

   a) We supply the `cons` function the arguments `1` and `2`.

   b) We supply the `cons` function the cons we just created and `3`.

   c) We supply the `setq` function the arguments `r` and the cons whose cdr is `3`.

      What is the printed representation of the value of `r`?

   d) We supply the `list` function with (in order) `4`, `5`, the value of `r`, and `nil`.
      What is the printed representation of this list?
      What is the length of this list (i.e., how many members are there)?

6. Convert the following expressions to prefix notation. Note that these require the arithmetic functions, such as `sqrt` (for finding square roots), `sin` (for finding sines), and `expt` (for performing exponentiation). These subrs are explained in Section 3.1.

   a) `area = PI * r`^{2}

   b) `circum = 2 * PI * radius`

   c) $r = \sqrt{x^2 + y^2 + z^2}$

   d) `d = (w * L`^{3}`)/(3 * e * i)`

   e) `p = (w * r * sin (phi + beta)) / a`

3

Arithmetic Functions

Topics Covered in This Chapter

- Arithmetic functions
- Number conversion
- Advanced topic: Bit manipulation functions

Goals for This Chapter

- Learn the contracts of the various arithmetic functions. Because we're already familiar with how most of these are used mathematically, they provide a friendly basis for exploring AutoLISP code concepts in the next part of this book.
- Gain an understanding of the bit manipulation functions, `lsh`, `logand`, `logior`, `~`, and `boole`. These esoteric functions are rarely explained and uncommonly employed but are quite useful in particular situations.

Introduction

The arithmetic functions allow us to perform common mathematical operations, such as addition and subtraction, find sines and cosines, and compare and convert numbers. They are familiar to most of us through our experience with mathematics or with other programming languages and are therefore suitable for demonstrating how the AutoLISP evaluation process works in the next chapter.

If you are new to AutoLISP, don't spend too much time puzzling over coding examples in this chapter; come back to them once you've completed Part II of the book.

3.1 Basic Arithmetic Functions

The arithmetic functions are a class of functions used to perform mathematical operations; in this section we see the ones that most programmers frequently use. The arithmetic functions work only on numbers, and many of them take as many arguments as we wish to give them.

In general, if *all* of the arguments to an arithmetic function are integers, then the result is returned as an integer. If *any* argument is a real number, then the result is a real number.

+ num . . . num	Returns the sum of successive additions.

```
Command: (+ 3 4)
7
```

As with most arithmetic functions, + takes an unlimited number of arguments. If all arguments are integers, the result is an integer:

```
Command: (+ 5 3 -2 7 13)
26
```

If *any* argument is a real number, then the result is a real number:

```
Command: (+ 5 3.0 -2 7 13)
26.0
```

All arguments *must* be numbers:

```
Command: (+ "65" 4)                    ;"65" is a string
error: bad argument type
```

We can use a symbol whose value is a number:

```
Command: (setq n 5)
5

Command: (+ n 6)
11
```

If supplied with only 1 argument, + returns that number:

```
Command: (+ 3)
3
```

1+ num	Takes a number and adds 1 to it.

```
Command: (1+ 3)
4

Command: (1+ -3.6)
-2.6

Command: (1+ 3 4)
error: too many arguments
```

1+ is useful for incrementing (increasing) a number, especially when iterating through a loop. It's faster than +, takes less space in memory, and less typing at the keyboard. But it doesn't do anything that + cannot do.

- num . . . num	Returns the remainder of successive subtractions from the first argument.

```
Command: (- 8 6)
2

Command: (- 25 3 -4 6)
20
```

In the second example, 3 is subtracted from 25, yielding 22. Then –4 is subtracted from 22, giving 26. Finally, 6 is subtracted from 26, returning 20.

If supplied with only one argument, - returns the negative of that number:

```
Command: (- 6)
-6
```

1- num	Takes a number and subtracts 1 from it.

```
Command: (1- -7)
-8

Command: (1- 3.8)
2.8
```

1- is useful for decrementing (decreasing) a number, especially when iterating through a loop. It's faster than -, takes less space in memory, and less typing at the keyboard. But it doesn't do anything that - cannot do.

* num . . . num	Returns the product of successive multiplications.

```
Command: (* -3 4)
-12

Command: (* 2 7.0 3)
42.0
```

If supplied with only one argument, * returns that number:

```
Command: (* 6)
6
```

/ num . . . num	Returns the quotient of successive divisions.

```
Command: (/ 12 3)
4

Command: (/ 64 4 8)
2
```

In the second example, 64 is divided by 4, yielding 16. Then 16 is divided by 8, returning 2.

If supplied with only one argument, / returns that number:

```
Command: (/ 9)
9
```

There are several dark corners that we must avoid when doing division:

- In any computer language (as well as in standard mathematics) it is illegal to divide by zero. Although we wouldn't do this intentionally, it is very easy to divide by a variable that has accidentally been set to zero:

```
Command: (setq x 0)
0

Command: (/ 9 x)
error: divide by zero
```

- Unlike addition or multiplication, the order of arguments is critical in division (and subtraction too). Reversing the order of two arguments causes reciprocal results to be returned:

```
Command: (/ 8.0 2)
4.0

Command: (/ 2 8.0)
0.25
```

- Results of integer division may differ radically from division of real numbers. Because the division of integers returns an integer, any fractional result is truncated and lost:

```
Command: (/ 8 5)
1

Command: (/ 8 5.0)
1.6

Command: (/ 2 3)
0
```

  The last example shows a perfect setup for a subsequent divide by zero error. Be careful! We can force a number to be a real with the `float` subr, which is shown in the following section.

Sometimes we want to perform an integer division even though the fractional precision is lost. In these cases the `rem` function is useful.

rem num . . . num	Returns the remainder of successive divisions.

rem is usually called with only two arguments. It divides the first argument by the second argument and returns the remainder:

```
Command: (rem 27 6)          ;(/ 27 6) gives a remainder of 3
3

Command: (rem 27.0 6)
3.0
```

On occasion we may need to perform multiple divisions:

```
Command: (rem 15 9 4)
2
```

In this example, 15 is divided by 9, yielding a remainder of 6. Then 6 is divided by 4, giving a remainder of 2.

gcd int int	Returns the greatest common denominator of two integers.

The *greatest common denominator* is the largest integer that divides evenly (i.e., with no remainder) into the two supplied integers. Both arguments must be integers.

```
Command: (gcd 64 24)
8

Command: (gcd 64 24.0)
error: bad argument type
```

If more than two arguments are supplied, the first two are used and the rest are ignored:

```
Command: (gcd 64 24 2 7 5)
8
```

expt num num	Returns the result of raising the first number to the power indicated by the second.

expt performs exponentiation in the following manner:

```
Command: (expt 2 3)          ;Raise 2 to the third power
8
```

Exponentiation can also be performed with the multiply function. The following example also raises 2 to the third power:

```
Command: (* 2 2 2)              ;Raise 2 to the third power
8
```

As a rule of thumb, when we raise numbers to the fourth (or higher) power, exponentiation is more efficient than multiplication.

We can use expt to find roots of a number:

```
Command: (expt 16 0.5)          ;4.0 is the square root of 16
4.0

Command: (expt 16 0.25)         ;2.0 is the fourth root of 16
2.0
```

expt can perform multiple exponentiations:

```
Command: (expt 2 3 4)
4096
```

In this example, 2 is raised to the third power, yielding 8. Then 8 is raised to the fourth power, giving a result of 4,096. This feature isn't reported in the *AutoLISP Reference.*

Be careful with expt; it is easily confused with the exp function, defined shortly.

sqrt num	Returns the square root of a number.

When we want to find the square root of a number, `sqrt` is more straightforward than `expt`:

```
Command: (sqrt 9)
3.0
```

`sqrt` is unusual in that even if its argument is an integer, the result is a real number. This is because the square root must be calculated using real numbers.

max num . . . num	Returns the largest (i.e., most positive) number.

```
Command: (max 8 -234 66 2)
66

Command: (max 3 5 1.0 2)
5.0
```

The first example returns the most positive number, 66, even though the absolute value of –234 is greater.

The second example returns a real result because one of its arguments is a real number. It doesn't matter that the largest number is specified as an integer.

min num . . . num	Returns the smallest (i.e., most negative) number.

```
Command: (min 0.7 1.4 0.25 1.1)
0.25

Command: (min 8 -2 0.6 9)
-2
```

The second example returns the most negative number, –2, even though 0.6 has a smaller absolute value.

abs num	Returns the absolute value of a number.

```
Command: (abs -23.53)
23.53
```

abs can be used with max or min when we don't know the sign of the arguments and we just want the greatest or smallest quantity, positive or negative.

```
Command: (setq x 5)
5

Command: (setq y -9)
-9

Command: (max (abs x) (abs y))
9
```

log num	Returns the natural log of a number.

```
Command: (log 2)
0.693147
```

exp num	Returns the natural antilog of a number.

The natural antilog is the value of *e* raised to the power of the argument. exp is used as follows:

```
Command: (exp 1)
2.71828
```

Be careful with exp; it is easily confused with expt !

♦ ♦ ♦

AutoLISP has several trigonometric functions. These functions expect their arguments to be expressed in radians rather than degrees. We use the functions dtr and rtd to convert degrees to radians and radians to degrees, respectively. These are not built-in AutoLISP functions; we must write them ourselves.

dtr, rtd, and radians are explained in Section 5.1.1.

sin num	Takes an angle expressed in radians and returns its sine.

```
Command: (sin 0.0)
0.0

Command: (sin (dtr 180))
1.0
```

cos num	Takes an angle expressed in radians and returns its cosine.

```
Command: (cos 0.0)
1.0

Command: (cos pi)
-1.0
```

There is no tan function in AutoLISP. However, a tangent can be computed by dividing the sine of an angle by the cosine of the same angle. We do this as an example in Section 5.1.1.

atan num1 [num2]	Returns the arctangent of a number in radians. If given a second argument, atan returns the arctangent of (/ num1 num2).

```
Command: (atan 1)
0.785398

Command: (atan -1)
-0.785398

Command: (rtd (atan -1))
-45.0

Command: (atan 2 3)
0.588003
```

If a second argument of 0 is supplied, atan returns ±90° (1.5708 radians), depending on the sign of the first argument.

```
Command: (atan 1 0)
1.5708

Command: (atan -5 0)
-1.5708
```

3.2 Number Conversion

When we perform arithmetic operations on integers, the result is usually an integer. There are some exceptions, such as `sqrt` and `cos`, which return a real result regardless of the numeric argument type. Arithmetic operations on real numbers always return a real result.

Sometimes we need to convert a real number into an integer or an integer into a real. For example, if we divide one integer by another, the result is an integer. This can lead to inaccuracies, as the following example shows:

```
Command: (/ 5 2)
2
```

When we divide two integers the result is an integer; the fractional portion is truncated. We can alleviate this problem by making one of the arguments be a real number:

```
Command: (/ 5 2.0)
2.5
```

It doesn't matter which argument is the real number, as long as at least one is.

Suppose we supply variables rather than actual numbers:

```
Command: (setq n1 8)
8

Command: (setq n2 5)
5

Command: (/ n1 n2)
1
```

We can't just put a ".0" at the end of the symbol name; `n1.0` has no meaning! Instead, we use the `float` function to force the number to be a floating point number.

`float` num	Takes a number and converts it into a real number.

```
Command: (float 3)
3.0
```

If the number is already real, `float` has no effect:

```
Command: (float 7.03)
7.03
```

We can supply variables to `float` (assume `n1` and `n2` have been set to 8 and 5 as before).

```
Command: (float n2)
5.0

Command: (/ n1 (float n2))
1.6

Command: !n2
5
```

Note that `n2`'s value is still the integer 5. A variable's value is not changed unless we `setq` it.

We can also convert a number from real to integer:

`fix` num	Takes a number and converts it into an integer.

```
Command: (fix 7.9879)
7
```

If the number is already an integer, `fix` has no effect:

```
Command: (fix -3)
-3
```

`fix` uses truncation to perform the conversion:

```
Command: (fix 5.9999)
5
```

There is no `round` subr in AutoLISP. If we need to round numbers, we must write this function ourselves. That task is presented as a lab at the end of Chapter 5.

3.3 Advanced Topic: Demystifying the Bit Manipulation Functions

Five arithmetic functions return results based on bits set in their integer arguments. They treat numbers as a collection of bits rather than as a magnitude.

Unless you have an immediate need for these functions, don't concern yourself with them until you know AutoLISP fairly well. To use them correctly you must understand binary numbers, bits, and Boolean algebra. These concepts are all explained in Section 14.2.

`lsh` int1 int2	Logically shifts the bits in int1 an amount indicated by int2.

`lsh` shifts the bits in its first argument an amount specified by its second argument. If int2 is positive, the bits in int1 are shifted to the left. If int2 is negative, the bits in int1 are shifted to the right. Bits shifted out of int1 are lost, and vacated bit positions are filled with zeros. The following examples illustrate the use of `lsh`:

```
Command: (lsh 5 1)
10
```

In this example, the number 5_{10} (0101_2) is shifted one digit to the left, returning the number 1010_2. This is equivalent to the decimal number 10.

```
Command: (lsh 5 -1)
2
```

Now 0101_2 is shifted one digit to the right. A 0 is introduced on the left and a 1 is lost on the right. The result is 0010_2, which is 2_{10}.

```
Command: (lsh -1 2)
-4
```

Negative numbers can also be shifted. In this example 11111111_2 is shifted 2 bits to the left. The returned value is 11111100_2, which is – 4 in base 10.

```
Command: (lsh 5.0 1)
error: function undefined for real
```

This example returns an error because one of its arguments is not an integer.

`logand` int . . . int	Performs a bit-by-bit logical AND on each of the integers.

The bitwise ANDing of integers returns a number representing the sum of the bits common to *all* of the arguments.

```
Command: (logand 13 7 5)
5
```

13 (1101_2) is composed of 8, 4, and 1; 7 (0111_2) consists of 4, 2, and 1; 5 (0101_2) comprises 4 and 1. Only the numbers 4 and 1 are common to all three arguments. They are added together and the result, 5, is returned.

```
Command: (logand 3 2.0)
error: function undefined for real
```

This example returns an error because one of its arguments is not an integer.

Here's an example of how we might use `logand`. Suppose we set several running snaps:

```
Command: osnap
Object snap modes: end, mid, int
```

There is an AutoCAD system variable named OSMODE that stores the current snap modes via bit values. For example, ENDPOINT is 1, MIDPOINT is 2, CENTER is 4, and INTERSECTION is 32. Instead of using the OSNAP command, we have the option of setting these snap modes directly into the OSMODE system variable:

```
Command: (setvar "osmode" 35)        ;Set END, MID, and INT
35
```

We can later learn which snap modes are set by examining this variable:

```
Command: (getvar "osmode")
35
```

Suppose our program needs to determine whether the MIDPOINT snap has been set. We can accomplish this with the `logand` function:

```
Command: (logand (getvar "osmode") 2)
2
```

Because the OSMODE bit value of 2 has been set for MIDPOINT, that bit is set in both arguments and returned by `logand`. On the other hand, CENTER has not been set, so `logand` returns 0:

```
Command: (logand (getvar "osmode") 4)
```

The setvar and getvar functions are explained in Section 5.2.4. See Section 11.3.1 for an example using logand to test whether a layer is frozen.

`logior` int . . . int	Performs a bit-by-bit logical Inclusive OR on each of the integers.

The bitwise inclusive ORing of integers returns a number representing the sum of the bits common to *any* of the arguments.

```
Command: (logior 9 3)
11
```

9 (1001_2) is composed of 8 and 1; 3 (0011_2) comprises 2 and 1. Thus, the numbers 8, 2, and 1 are present in at least one of the arguments. These numbers are added together, and the result, 11, is returned.

```
Command: (logior -8 12)
-4
```

–8 is 11111000_2. 12 is 00001100_2. When added they return $11111100_2 = -4_{10}$.

```
Command: (logior 9 3.0)
error: function undefined for real
```

This example returns an error because one of its arguments is not an integer.

~ int	Returns the bitwise NOT of an integer.

~ flips every bit in the supplied argument so that each 1 becomes 0 and each 0 becomes 1.

```
Command: (~ 12)
-13
```

Flipping the bits for $12_{10} = 001100_2$ yields $110011_2 = -13_{10}$.

```
Command: (~ -4)
3
```

Flipping the bits for $-4_{10} = 11111100_2$ yields $00000011_2 = 3_{10}$.

```
Command:(~ 12.4)
error: bad argument type
```

This example returns an error because its argument is not an integer.

`boole` bfun int . . . int	A general function for operating on bits in numbers. It performs a bit-by-bit comparison of the integers according to bfun, a specified Boolean function.

At best, `boole` is a difficult function to comprehend the first time through. To complicate matters further, several terms sound alike but refer to different components of this function:

`boole` function	The name of this AutoLISP subr
boolean function (bfun)	The first argument to the `boole` function
function bits	Bits that are set as the result of the Boolean comparison of the second and third (and remaining) arguments

For the most part we'll examine Boolean operations on two integers only, although we will see an example of Boolean operations on multiple integers.

The `boole` function begins by comparing each corresponding bit in its second and third arguments. In other words, the first bit of int1 is compared with the first bit of int2, the second bit of int1 is compared with the second bit of int2, and so forth. When two bits are compared, there are four possibilities:

- They are both 0
- The first is 0 and the second is 1

- The first is 1 and the second is 0
- They are both 1

The result of this comparison causes a function bit to be set for each bit position according to the following chart:

int1	*int2*	*function bit*
0	0	8
0	1	4
1	0	2
1	1	1

Let's look at an example of how the function bits get set. Compare the number 2 with the number 3:

```
2 =                          10
3 = 2 + 1 =                  11
Function bits returned =     14
```

The first bit of each is 1, causing the first function bit to be set to 1. The second bit of 2 is 0 and the second bit of 3 is 1, which causes the second function bit to be set to 4.

"So what?" you may ask. Be patient; this is a very complex function that must be understood in steps.

We next compare each function bit with the bfun supplied as the first argument to `boole`. The bfun is a positive integer in the range 0 to 15. If the function bit matches a 1 in the bfun, then that bit is set to 1 in the result. If there is no match, then that bit is set to 0 in the result.

We found that a comparison of the integers 2 and 3 caused function bits 1 and 4 to be set. Let's see how these function bits interact with different settings of the bfun. First we'll set the bfun to 1:

```
Command: (boole 1 2 3)
2
```

The first function bit, 1, matches the bfun bit, so a result of 1 is given. The second function bit, 4, does not match the bfun bit, so a result of 0 is given for the second bit. Thus the entire result is 10_2 or 2_{10}.

Next we'll change the bfun to 7 but keep the function bits the same. The binary representation of 7 is 111_2, which means that this bfun will match function bits 4, 2, and 1.

```
Command: (boole 7 2 3)
3
```

The first function bit, 1, matches one of the bfun bits, so a result of 1 is given for the first bit. The second function bit, 4, also matches a bfun bit, so a result of 1 is given for the second bit as well. Thus the entire result is 11_2 or 3_{10}.

Several bfun combinations are particularly useful and are summarized in the following chart:

bfun	*operation*	*Result bit is 1 if*
1	AND	Both input bits are 1
6	XOR	Only one input bit is 1
7	IOR	Either or both input bits are 1
8	NOT	Both input bits are 0

Let's see how the function bits are used with the bfun AND and XOR combinations:

- For an AND to return 1, all of its arguments must return 1.

  If the two bits being compared are both 1, then function bit 1 is set. It matches the bfun 1, so 1 is returned.

  If either compared bit returns 0, then one of the other function bits is set. None of them match the bfun 1, so 0 is returned for that bit position.

- For Exclusive OR (XOR) to return 1, one but not all of its arguments must return 1.

  If the first bit being compared is 1 and the second is 0, then function bit 2 is set. If the first bit being compared is 0 and the second is 1, then function bit 4 is set. Either case will match the bfun 6 (4 + 2) and return 1.

  If both compared bits are 1, then function bit 1 is set. If both compared bits are 0, then function bit 8 is set. Neither matches the bfun 6, so 0 is returned for that bit position.

If more than two arguments are being compared, then an odd number of arguments must return 1.

Let's look at a few examples of `boole` usage. For clarity, we'll use the same numbers for comparison but change the bfun.

Example 1

Boolean function 1 (AND) returns 1 if and only if both compared bits are 1.

```
Command: (boole 1 22 17)
16

bfun = 1
22 = 16 + 4 + 2 =          10110
17 = 16 + 1 =              10001

Function bits returned =   18224

Matches                    10000
```

Each bit of 22 is compared with each bit of 17. These return the function bits 1,8,2,2,4. Each function bit is then compared to the bfun, 1. Only the 16 bit matches, putting a 1 in the first position and 0s in the remaining spots. Thus the `boole` function returns $10000_2 = 16_{10}$.

Example 2

Boolean function 6 (XOR) returns 1 if the first bit is 1 and the second is 0, or if the first bit is 0 and the second is 1. It returns 0 if both bits are 0 or if both bits are 1.

```
Command: (boole 6 22 17)
7

bfun = 6 = 4 + 2
22 = 16 + 4 + 2 =          10110
17 = 16 + 1 =              10001

Function bits returned =   18224

Matches                    00111
```

Each bit of 22 is compared with each bit of 17. These return the function bits 1,8,2,2,4. Each function bit is compared with the bfun bits, 4 and 2. The last three bits match one of these. Therefore, the `boole` function returns 7 (4 + 2 + 1).

The examples we have seen so far have used two numbers to determine the function bits. When we provide the `boole` function with three numbers for function bits, it initially performs its operation on the first two, then runs a second time with the result and the third number as arguments. Since we just ran the `boole` function once, let's call it again with the result, 7, and the number 4:

```
Command: (boole 6 7 4)
3
```

```
bfun = 6 = 4 + 2
7 = 4 + 2 + 1 =          111
4 =                      100

Function bits returned =  122

Matches                   011
```

Each bit of 7 is compared with each bit of 4, returning the function bits 1,2,2. Each function bit is then compared to the bfun bits, 4 and 2. The last two bits match the 2. Therefore, the `boole` function returns 3 (2 + 1).

Rather than doing this process in two steps, all three numbers can be handled by a single call to `boole`.

```
Command: (boole 6 22 17 4)
3
```

Example 3

Boolean function 8 (NOT) returns 1 if and only if both bits are 0. But its results are somewhat different from the Boolean functions shown previously because of the way numbers are stored in the computer. Since a number is stored in a 16-bit or 32-bit word, a positive integer usually has quite a few leading zeros. When the binary representation of a positive integer is presented, the leading zeros are typically omitted.

NOT flips bits, producing many leading 1s at the start of the resultant number. If the very first bit in a computer word is 1, then the number it represents is negative. Thus, many of the results returned by bfun 8 are negative numbers.

```
Command: (boole 8 22 17)
-24
```

```
bfun = 8
22 = 16 + 4 + 2 =          00010110
17 = 16 + 1 =              00010001

Function bits returned =   88818224

Matches                    11101000
```

Each bit of 22 is compared with each bit of 17. These return the function bits 1,8,2,2,4, with several leading 8s. Each function bit is then compared to the bfun, 8. The 8 bit and all the leading 8s match. Thus this function returns 11101000_2, which is the decimal number –24.

Notes

- All arguments to the `boole` function must be integers:

```
Command: (boole 2 22.4 17)
error: bad argument type
```

- The first argument to the `boole` function must be in the range 0 to 15 inclusive.

```
Command: (boole 22 17 4)
error: boole arg1 < 0 or > 15
```

♦ ♦ ♦

Now that you understand how `boole` works, you might be asking yourself, "Why in the world would we ever use it?" Most of `boole`'s potential uses involve an AND, OR, or NOT application. In those situations, `logand`, `logior`, and `~` are much simpler to use. `boole` comes in handy when we need to perform an operation that is not one of these three, such as XOR.

XOR has a unique property. When we apply XOR to two numbers, represented by `a` and `b`, we get a result, `c`. If we then apply XOR to `a` and `c`, we get `b` back again. Recall Example 2:

```
Command: (boole 6 22 17)
7
```

Let's run `boole` once more, using an original argument, 22, and the result, 7:

```
Command: (boole 6 22 7)
17
```

The other original argument, 17, is returned. If we supply 17 and 7, we get 22:

```
Command: (boole 6 17 7)
22
```

This property of XOR makes it an excellent tool for cryptography. We can XOR a key to a numeric value, causing that value to be encrypted. When we apply the same key to the encrypted number, we obtain the original value. Since we can easily convert characters to their numeric ASCII values, this is a great mechanism for encrypting files.

3.4 Important Points to Remember

- Arithmetic functions work only on numbers. Giving them something other than numbers will cause a "bad function" error. `"37"` is a string, not a number.
- When typing a number that is less than 1, remember to include the leading 0. Entering .6 will cause an "invalid dotted pair" error.
- Arithmetic functions performed on integers typically return an integer result. This can be dangerous with division:

  ```
  Command: (/ 7 4)
  0
  ```

  When performing division, make sure that at least one argument is a real number. The `float` function is one way to ensure that this will happen.
- Entering `(float n)` or `(+ 3 n)` doesn't change the value of `n`. To do so we must enter `(setq n (float n))` or `(setq n (+ 3 n))`.
- Don't confuse `exp` with `expt`. The former returns a natural antilog of a number; the latter performs exponentiation.

3.5 Labs

1. Enter each of the following AutoLISP functions at the Command: prompt and see what gets returned. Write down what you believe will be returned before you enter each one. Recall that `setq` is the function we use to assign a value to a variable, and that `!a` typed at the Command: prompt returns the value of the variable `a`.

```
Command: (setq a (+ 2 3))

Command: !a

Command: (setq b (* a 7))

Command: (setq c (1+ b))

Command: !b

Command: (setq b (sqrt 1+ b))

Command: (setq b (sqrt (1+ b)))

Command: !b

Command: (setq d (fix b))

Command: !b

Command: (setq e (/ c 2))

Command: (gcd c e)

Command: (gcd b e)

Command: (setq f (/ e 4))

Command: (setq g (/ e 4.0))

Command: (setq h (/ (float e) 4))

Command: (rem e 4)

Command: (max a c e 3)

Command: (max a b c e 3)

Command: (min a -8 f 6.0)
```

2. If you have read the Advanced Topic on bit manipulation functions, determine whether the following statement is true or false, and why:

   The expression `(lsh int1 int2)` is equivalent to the expression `(* int1 (expt 2 int2))`.

3. If you are an experienced AutoLISP user, you might want to do Exercises 2 to 6 in Chapter 5 at this time. They will give you practice writing functions that require the subrs introduced in this chapter.

II

Understanding How Code Is Executed in AutoLISP

Part II is the most vital portion of *Essential AutoLISP.* In these pages we learn how to write functions and how AutoLISP actually executes them.

Chapter 4 is devoted to the most important function in AutoLISP, `eval`. `eval` is the function that executes all of the code that we type at the Command: prompt or load in from a file. Understanding `eval` is *essential* for being able to program effectively in AutoLISP. Why is this?

- `eval` alters most of our code before executing it. Thus, the code that we supply to AutoLISP is often *not* the code that is actually run. What looks good on paper very often yields confusing error messages or runs on incorrect data. Once we understand `eval`'s process, we can anticipate its actions and avoid those errors.

- Most error messages we receive come from `eval` or from functions called by `eval`. Until we understand `eval`, messages like "bad formal argument list" have no meaning. When we know how these errors arise we can correct them instantaneously. 90 percent of the coding errors we make are somehow connected to `eval`.

To better understand `eval` I've provided an Eval Flowchart program. This is an on-line tutorial that prompts us to type in an AutoLISP expression, then shows the step-by-step actions taken by `eval` as it processes that expression. The examples and exercises will help you gain a deep understanding of the tasks performed by `eval`.

In Chapter 5 we learn how to write and run our own AutoLISP functions. We see how to use `defun` to write functions, then practice doing so. We learn another extremely important function, named `quote`, whose task it is to inhibit `eval` from running. The fact that we use `quote` in nearly every expression we write points out again the importance of understanding `eval`: If we don't know what it does, we won't know when *not* to use it!

One of the main features of AutoLISP is our ability to create our own AutoCAD commands that are virtually indistinguishable from the built-in ones. In Chapter 5 we learn how to write our own AutoCAD commands as well as how to run the standard AutoCAD commands from within AutoLISP.

4

Evaluation of AutoLISP Expressions

Topics Covered in This Chapter

- The `eval` function and the `eval` flowchart
- The conversion of the code we write into the code AutoLISP runs
- Evaluation of AutoLISP forms
- Errors encountered during evaluation
- The Eval Flowchart on-line tutorial

Goals for This Chapter

This chapter initiates our study of how AutoLISP manipulates code. We learn how our code is actually executed and, in the next chapter, how to write our own programs. Once we have thoroughly assimilated the concepts contained in this part of the book, we will have a solid foundation for programming in AutoLISP.

The skills gained in this chapter include the following:

- Understand all the steps taken by `eval`.
- Be able to parse (pull apart) and understand an arbitrarily complex piece of code.
- Understand why certain errors arise during the evaluation process and how to avoid and correct them.
- Learn to use the Eval Flowchart program to gain a better understanding of `eval`'s actions and its associated error messages.

Introduction

Problem solving involves modeling the real world inside the computer. We perform this familiar process all the time in our drawings, knowing full well that we're looking at, say, a wireframe, rather than a real object.

We can use AutoLISP to model the real world in the following ways:

- As a command processor for creating and running our own AutoCAD commands.
- As a calculator for performing mathematical operations.
- As a natural language translator to translate, say, a graphic language. Whereas a layman would think of a line as just a connection between two points, a surveyor might think of it as a *bearing*. AutoLISP allows surveyors to think in terms of bearings instead of simple lines.
- As a drafting system front-end. For example, let's say a surveyor would like to run a pavement line down a roadway. Instead of simply using the OFFSET command, she might write a function called `pavement-line`, which would offset lines and place them on different layers.

In AutoLISP, the various datatypes that we have seen—symbols, lists, and so forth—are our tools for modeling real world objects inside of the computer. Not only can we use AutoLISP objects to model their real-world counterparts, but we can also use the same object types to model *actions* within AutoLISP. In this chapter we learn how AutoLISP performs these actions.

Almost everything we do in AutoLISP is accomplished by applying a function to arguments. For example, the code for adding the numbers 3 and 4 together consists of two parts:

1. the function name: +
2. the arguments to the function: 3 and 4.

Figure 4-1 shows how we can model this form as a list:

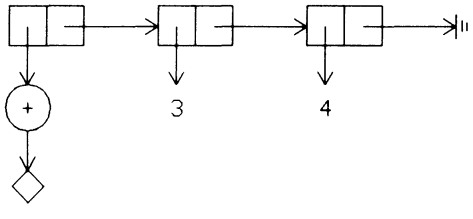


Figure 4-1

The printed representation of this list is `(+ 3 4)`.

This brings us to the function `eval`. The way we accomplish tasks in AutoLISP is to create a model of an operation and hand it to `eval`.

4.1 How the eval Function Processes Our Code

`eval` has a very complex contract; it is the heart of AutoLISP. Once we completely understand `eval`, the AutoLISP language fits together clearly and logically. Until we understand `eval`, however, we'll always be guessing what the machine is going to do.

In some ways `eval` is just a function like any other function. We can call it like we call the `car` function, in which case it runs and returns a result. However, it does much more than any other function and has many special characteristics. One feature is that `eval` automatically processes every AutoLISP expression we type at the Command: prompt; it is the engine that powers the language. Since `eval`'s actions are transparent, many AutoLISP users aren't even aware of its existence. Let's study `eval`'s contract.

`eval` takes one argument, which we call a form, and performs a process called *evaluation*. A *form* is *any* AutoLISP object that is supplied to `eval`. It can be a number, symbol, list, or any other AutoLISP object. `eval` is such a special function that its argument and its process are given names.

What does `eval` do with this form? That depends on what kind of object the argument is.

- If the argument is *anything* other than a symbol or a cons, then `eval` returns the object untouched.

  For example, if we give `eval` the number 3, then `eval` returns the number 3. If we give `eval` the string `"Height = 7.5"`, then `eval` simply returns that string. In these cases `eval` is what we call a "no-op"; it performs no operation.

- If the argument is a symbol, `eval` returns the *value* of that symbol. By default, a symbol's value is `nil`.

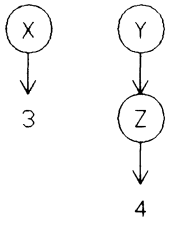


Figure 4-2

Consider Figure 4-2. If we hand the symbol `x` to `eval`, then `eval` returns its value, 3. If we hand the symbol `y` to `eval`, then `eval` returns its value, `z`. Note that `eval` does not return 4 in the latter case because 4 is the value of `z`, not `y`, and we never trace through arrows. The arrow coming from a symbol always points directly to the symbol's value. If we have `y` and want 4, we must hand `y` to `eval` to get `z`, then hand `z` to `eval` to get 4. It takes two steps to move along two arrows.

There is a "handshaking" that goes on between `eval` and `setq`. We use `setq` to assign a value to a symbol. Then, when we give that symbol to `eval`, `eval` returns the symbol's value. In some sense we don't care about symbols; they're just objects that are used to name or reference data that we do care about. We often use `setq` to label an object that does not have a name, such as a list. The list itself is the object that interests us, not the symbol bound to it. When we give the symbol to `eval`, it evaluates the symbol and returns the list.

- If the argument is a cons, then that cons *must* be a valid list.

The following flowchart depicts the steps we've seen so far:

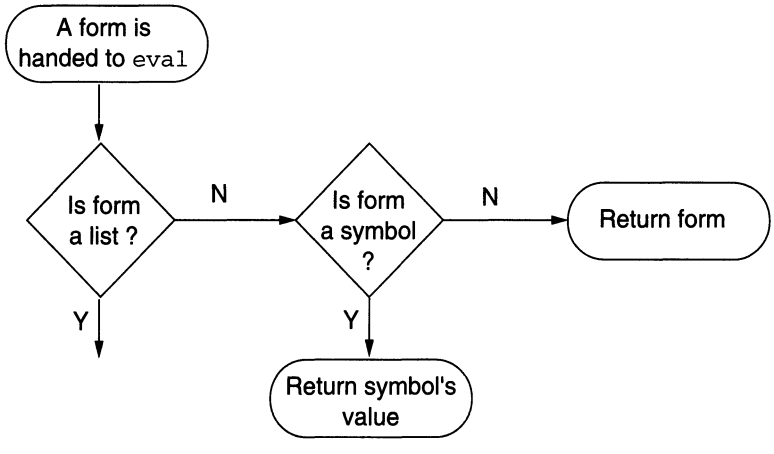


Figure 4-3

4.1.1 How `eval` Handles Lists

When `eval` is handed a form that is a list, its contract is more complex:

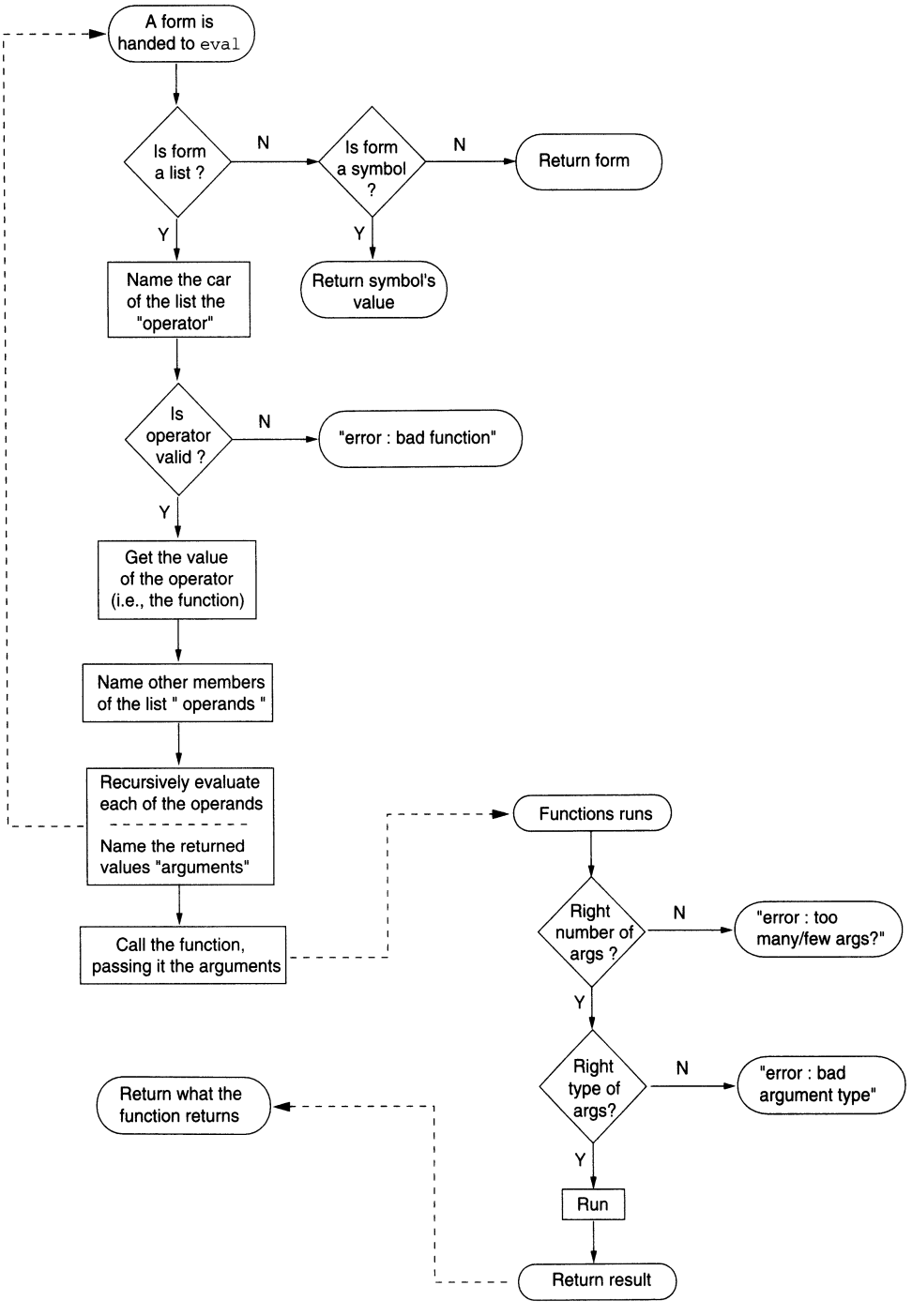

A form is handed to eval
Is form a list ?
N
Is form a symbol ?
N
Return form
Y
Y
Name the car of the list the "operator"
Return symbol's value
Is operator valid ?
N
"error : bad function"
Y
Get the value of the operator (i.e., the function)
Name other members of the list " operands "
Recursively evaluate each of the operands
Name the returned values "arguments"
Call the function, passing it the arguments
Functions runs
Right number of args ?
N
"error : too many/few args?"
Y
Right type of args?
N
"error : bad argument type"
Y
Run
Return result
Return what the function returns

Figure 4-4

eval's steps are worth examining in detail. We'll use the following example for illustration. The sqrt function, you'll recall, finds the square root of a number.

```
Command: (sqrt 25)
5
```

1. Name the car of the list the *operator.* The operator *must* be a symbol whose value is a function. In this example the operator is the symbol sqrt.

2. Get the value of the operator, that is, the square root function, and set it aside for later. At this point eval doesn't need the function, but it must verify that it does indeed have a function. If the operator is not a symbol or if the value of the operator is not a function, then AutoLISP will issue an error. The possible error messages are discussed in the following section.

3. Name the other members of the list *operands.* This example has only one operand, the number 25. (Recall that the members of a list are the cars of the backbone conses. In a list of length *n,* the first element is the operator and the remaining *n–1* elements are operands.)

4. Hand each operand in turn to eval for evaluation. eval is a *recursive* function. If you are new to programming, it is important that you read and understand function calling and recursion before proceeding further. (Function calling and recursion are explained in Section 14.3.2.)

   Even experienced programmers might have difficulty keeping straight the many levels of function calls that eval makes to itself. To help clarify the recursion process, we will pretend that instead of having one eval function that calls itself many times, we have several identical copies of eval that get called in turn.

   In Section 14.3.2 we discuss how function f1 would call f2, f2 would call f3, and so forth. The model we use is shown in Figure 4-5:

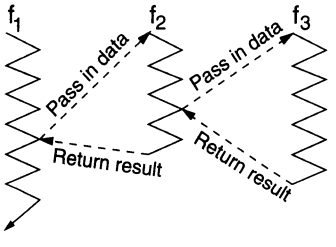


Figure 4-5

We will use a similar model, shown in Figure 4-6, for eval's recursive calls:

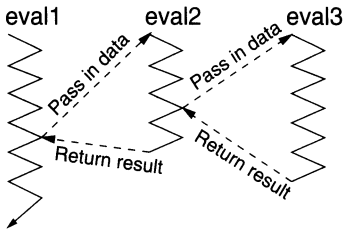


Figure 4-6

The first time eval is called, we'll refer to it as "eval Level 1" or "eval1." When eval1 calls itself, we'll say it's calling eval Level 2. eval2 calls eval Level 3, and so forth. In this manner we'll ignore eval's recursion and pretend that independent functions are being called. This practice in no way inhibits us and makes eval's actions easier to understand. Keep in mind, however, that there is only one eval function.

When eval2 calls eval3, it waits at the spot where it issued that function call until a result comes back from eval3. When eval3 finishes running, it returns a result to eval2, because a function always returns its result to the function that called it.

5. eval1 calls eval2 once for each operand. The results that are returned are called *arguments*. In Chapter 1 we defined arguments to be the data supplied to functions. Here's another, equally important, definition:

An *argument* is an evaluated operand.

eval calls itself recursively once for each operand, until all of the operands have been evaluated and it has the arguments. In our example, the sole operand is the number 25. When we hand a number to eval, eval simply returns that number. In this case the argument is identical to the operand, but in many instances it will differ.

6. Call the function, and pass it the arguments. In this example we give the argument 25 to the square root function. eval waits while the function performs the following steps (this is yet another function call):

   a) Checks to see whether it received the right number of arguments. If not, it issues one of the following errors and aborts execution:

"too many arguments"
"too few arguments"
"incorrect number of arguments to a function"

The square root function requires one argument and receives one argument, so it's happy.

b) Checks to see whether it received the right type of arguments. If not, it issues a "bad argument type" error. Square root requires a number and receives a number, so it's very happy.

c) If we fulfill our end of the function's contract by supplying the right number and right type of arguments, then the function performs the task that we called it to do. Our function finds the square root of 25.

d) Returns a result. Every function returns a result (unless it encounters an error). Square root returns its result, 5, back to `eval`.

Some functions, such as `car` or `sqrt`, are information-gathering functions. Their purpose is to tell us something about the AutoLISP world. The result they return is the information that we seek.

Other functions, such as `cons`, change the AutoLISP world, usually by creating new objects or, as with `setq`, changing the relationships among existing objects. These functions also return a result, and we can use their results as input to other functions.

7. When `eval` receives the result from the function, it sends that result back to whoever made the original call to `eval`. This might be the user, a function, or the next higher level of `eval` recursion. Since we typed `(sqrt 25)` at the Command: prompt, the result, 5, is printed on our terminal screen.

4.1.2 Errors Reported by `eval`

Several different errors can arise during the evaluation process. These are among the most common errors that AutoLISP users encounter. Once we understand the cause of each error, we can quickly debug our code.

When an error is encountered, processing is immediately aborted and an error message is printed on the screen. If the error occurs within a function, the message is *not* returned to us through `eval`.

The first argument to `eval` must be an operator, that is, a symbol whose value is a function. If presented with something other than a valid operator, `eval` issues an error message. Suppose we enter the following list at the Command: prompt:

```
Command: (a b c)
```

One of four events may occur:

1. The symbol `a` is the name of a function that we defined. In this case the `a` function is executed properly and no error message ensues. Inasmuch as `a` is not a very meaningful function name, this option is not probable.

2. The value of `a` is an atom, such as a number, symbol, or string. Suppose we had earlier entered:

   ```
   Command: (setq a 1)
   1
   ```

   Now `a`'s value is a number, not a function, and `eval` issues a "bad function" error:

   ```
   Command: (a b c)
   error: bad function
   ```

3. The value of `a` is `nil`. It is highly likely that we never assigned a value to `a`, so it is unbound. In this case `eval` issues a "null function" error. "null function" is a specific case of "bad function," where the operator's value is `nil` as opposed to being a number, string, or other AutoLISP object.

4. The value of `a` is a list or a cons. Suppose we had earlier entered one of the following forms:

   ```
   Command: (setq a (list 1 2 3))
   (1 2 3)
   ```

   or

   ```
   Command: (setq a (cons 1 2))
   (1 . 2)
   ```

   Now `a`'s value is a list or a cons, not a function, and `eval` issues the following error message:

   ```
   Command: (a b c)
   error: bad formal argument list
   ```

"bad formal argument list" is a derivative case of "bad function" where the operator's value is specifically a list or cons.

Note: There are numerous other ways in which these errors might arise. If instead of entering `(+ 5 7)` we enter `(5 + 7)` or `(+ (5 7))`, we will also get a "bad function" error.

When AutoLISP prints an error message, it always follows the message with a listing of the code that caused the error. Although there may be several lines of code, the first line after the error message *always* shows the exact place where the error occurred. For example, let's enter the following erroneous form:

```
Command: (* 2 (a b c))
error: null function
(a b c)
(* 2 (a b c))
```

By examining the first line following the message, we can immediately discern that `a` is the name of the invalid operator. The second line merely displays the larger piece of code that contains the offending form.

See Section 12.1 for further discussion of these and other error messages.

4.1.3 Examples

The following examples "walk through" the `eval` flowchart for various AutoLISP forms. Each one shows a different aspect of the evaluation process. You are encouraged to study the `eval` flowchart as you read these examples and practice going through `eval`'s steps until you know them extremely well.

To help clarify the recursion process:

⇒	An arrow indicates actions taken by `eval` Level 2.
⇒	An indented arrow indicates actions taken by `eval` Level 3.
◊	A diamond indicates actions performed by a function.

Remember, however, that there is only one `eval` function. The `eval` levels are merely used to indicate and clarify the current layer of recursion.

In general, when you get stuck on a problem or do not understand why you received a particular error message, step through the flowchart with the problem code. Keep in mind that the flowchart is a pictorial representation of the exact steps taken by AutoLISP.

Example 1

Let's begin with a form we saw earlier:

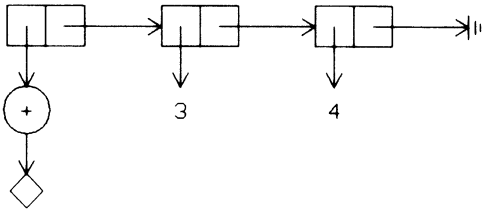


Figure 4-7

A form is handed to `eval` Level 1. The form is `(+ 3 4)`.

Is this form a list? Yes

Name the car of the list the "operator." The symbol + is the operator.

Get the value of the operator and save it for later on. This is the addition function.

Name the other members of the list "operands." The numbers 3 and 4 are the operands.

Recursively evaluate each of the operands. Hand the first operand, 3, to `eval` Level 2 for evaluation. `eval1` waits at this spot until a result is returned while `eval2` commences at the start of the flowchart.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** No

⇒ **Return the result to `eval1`.** The result is 3.

Hand the second operand, 4, to `eval2` for evaluation. `eval1` waits at this spot until a result is returned.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** No

⇒ **Return the result to `eval1`.** The result is 4.

`eval1` now has its two arguments, 3 and 4. In this example the operands and arguments are the same; we'll soon see examples where they are different.

Call the addition function, passing it the arguments, 3 and 4. The addition function performs the following steps:

- ◊ **Are there the right number of arguments?** Addition doesn't care how many arguments it is given.
- ◊ **Are these the right type of arguments?** Addition needs numbers and gets numbers.
- ◊ **Add the numbers.**
- ◊ **Return the result back to `eval1`.** The result is 7.

`eval1` returns the result. 7 is printed on the screen.

Example 2

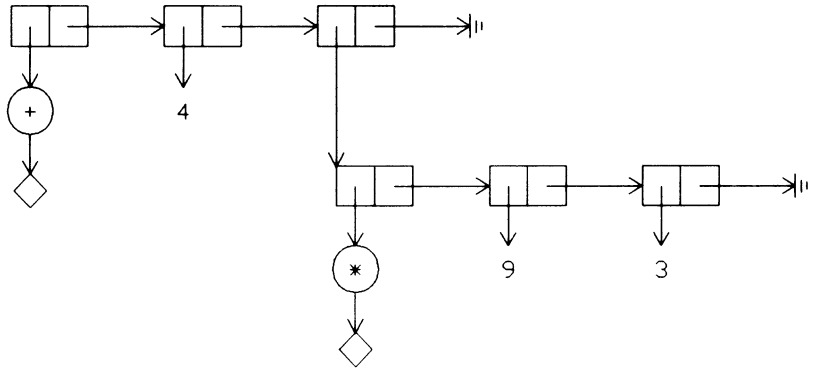


Figure 4-8

Before we start, what do you expect the evaluation of this expression, `(+ 4 (* 9 3))`, to return? How did you arrive at this answer?

The innermost parentheses are executed first. 9 is multiplied by 3 to get 27; this result is added to 4, yielding 31. Enter this form at the Command: prompt to verify that 31 is indeed returned.

`eval` isn't as smart as we are; it can't scan to locate the innermost parentheses. Instead, it goes left to right and reads this expression as, "I need to perform an

operation on 4 and . . . wait a minute: there's another list here! I'd better call myself to determine the second argument."

`eval` Level 2 receives the form `(* 9 3)` and says, "I need to perform an operation on 9 and 3." It hands these numbers to multiplication, gets 27, and returns that result back up to `eval1`. `eval1` now has its arguments, 4 and 27, and calls the addition function to add them together.

The entire `eval` flowchart (and code) has been devised to handle this situation. We frequently nest functions several levels deep and need to execute what's in the innermost parentheses first. `eval` scans left to right and calls itself whenever one of the operands is a list. When it gets down to where the operands are atoms, `eval` takes the path where it doesn't call itself. This causes `eval` to break out of the recursion.

Now let's examine the exact path that `eval` takes:

A form is handed to `eval` Level 1. The form is `(+ 4 (* 9 3))`.

Is this form a list? Yes

Name the car of the list the "operator." The symbol + is the operator.

Get the value of the operator and save it for later on. This is the addition function.

Name the other members of the list "operands." The number 4 and the list `(* 9 3)` are the operands.

Recursively evaluate each of the operands. Hand the first operand, 4, to `eval` Level 2 for evaluation. `eval1` waits at this spot until a result is returned, and `eval2` starts at the top.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** No

⇒ **Return the result to `eval1`.** The result is 4.

Hand the second operand to `eval2` for evaluation. This is `(* 9 3)`. `eval1` again waits at this spot while `eval2` runs.

⇒ **Is this form a list?** Yes it is, so `eval2` takes the "list" path of the flowchart.

⇒ **Name the car of the list the "operator."** The symbol `*` is the operator.

⇒ **Get the value of the operator and save it for later on.** This is the multiplication function.

⇒ **Name the other members of the list "operands."** The numbers 9 and 3 are the operands.

⇒ **Recursively evaluate each of the operands.** Hand the first operand, 9, to `eval` Level 3 for evaluation. `eval2` waits at this spot until a result is returned while `eval3` runs from the top.

  ⇒ **Is this form a list?** No

  ⇒ **Is this form a symbol?** No

  ⇒ **Return the result to `eval2`.** The result is 9.

⇒ **Hand the second operand, 3, to `eval3` for evaluation.** `eval2` again waits at this spot while `eval3` runs.

  ⇒ **Is this form a list?** No

  ⇒ **Is this form a symbol?** No

  ⇒ **Return the result to `eval2`.** The result is 3.

⇒ `eval2` now has its two arguments, 9 and 3.

⇒ **Call the multiplication function, passing it the arguments, 9 and 3.** The multiplication function performs the following steps:

  ◊ **Are there the right number of arguments?** Multiplication doesn't care how many arguments it is given.

  ◊ **Are these the right type of arguments?** Multiplication needs numbers and gets numbers.

  ◊ **Multiply the numbers.**

  ◊ **Return the result back to `eval2`.** The result is 27.

⇒ **`eval2` returns the result, 27, back to `eval1`.**

`eval1` now has its arguments, 4 and 27.

Call the addition function, passing it the arguments, 4 and 27. The addition function performs the following steps:

◇ **Are there the right number of arguments?** Addition doesn't care how many arguments it is given.

◇ **Are these the right type of arguments?** Addition needs numbers and gets numbers.

◇ **Add the numbers.**

◇ **Return the result back to `eval1`.** The result is 31.

`eval1` returns the result. 31 is printed on the screen.

Example 3

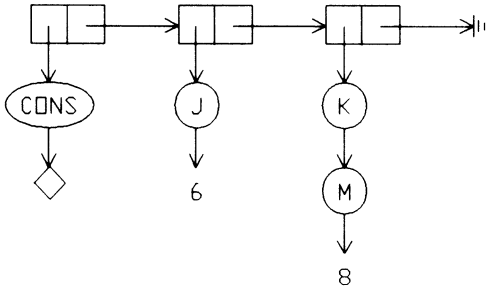


Figure 4-9

A form is handed to eval Level 1. The form is `(cons j k)`. Remember, the value of a symbol doesn't appear in the symbol's printed representation.

Is this form a list? Yes

Name the car of the list the "operator." The symbol `cons` is the operator.

Get the value of the operator and save it for later on. This is the `cons` function.

Name the other members of the list "operands." The symbols `j` and `k` are the operands.

Recursively evaluate each of the operands. Hand the first operand, `j`, to `eval2`

for evaluation. `eval1` waits at this spot until a result is returned.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** Yes

⇒ **`eval2` returns `j`'s value to `eval1`.** The result is 6.

Hand the second operand, `k`, to `eval2` for evaluation. `eval1` again waits at this spot while `eval2` runs from the top.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** Yes

⇒ **`eval2` returns `k`'s value to `eval1`.** The result is `m`, not 8, because `m` is the value of `k`. We don't trace through arrows.

`eval1` now has its two arguments, 6 and `m`.

Call the `cons` function, passing it the arguments, 6 and `m`. The `cons` function performs the following steps:

◊ **Are there the right number of arguments?** `cons` needs two arguments and that is what it is given.

◊ **Are these the right type of arguments?** `cons` doesn't care what type of arguments it receives.

◊ **Create a new cons with a car of 6 and a cdr of `m`:**

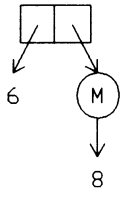


Figure 4-10

◊ **Return this new cons back to** `eval1`.

`eval1` returns the result. `(6 . M)` is printed on the screen. What happens to the

8 in this example? Nothing. It's still the value of `m`, but `m` isn't evaluated and the 8 isn't used.

Important Note

- Examine the form we just evaluated:

```
Command: (cons j k)
(6 . M)
```

  At the Command: prompt we entered the operator, `cons`, and operands, `j` and `k`. This conforms to the prefix notation rules.

  The value of the operator, which is the function, runs on the values of the operands, which are the arguments. In other words, we always supply AutoLISP with operators and operands, but ultimately functions run on arguments. Thus, what we see is *not* what we get! Figure 4-11 displays these relationships:

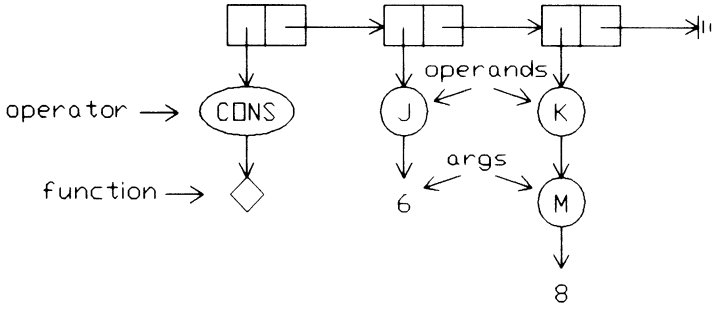


Figure 4-11

- When we write functions, we must always remember that what we supply to AutoLISP is transformed and that something different is usually run. It doesn't matter what *operands* we provide to a function; the *arguments* are what the function receives. In fact, when we study the contracts of the various subrs, we always examine the arguments those subrs take; we don't care what the operands are. There is no such thing as a "bad *operand* type" error.

  This is one of the biggest "gotchas" for new AutoLISP users. As we begin to program we must constantly remind ourselves to look below the surface of the code to determine the data that the function actually uses.

Example 4

At first glance this example appears to be nearly identical to the last. But we now know that we must look below the surface to see what really happens. Thus, it's very worthwhile going through `eval`'s steps for this expression.

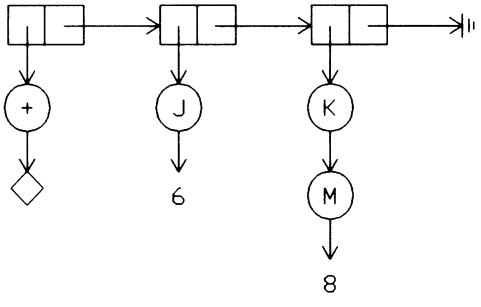


Figure 4-12

A form is handed to `eval` Level 1. The form is `(+ j k)`.

Is this form a list? Yes

Name the car of the list the "operator." The symbol + is the operator.

Get the value of the operator and save it for later on. This is the addition function.

Name the other members of the list "operands." The symbols `j` and `k` are the operands.

Recursively evaluate each of the operands. Hand the first operand, `j`, to `eval2` for evaluation. `eval1` waits at this spot until a result is returned.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** Yes

⇒ **`eval2` returns `j`'s value to `eval1`.** The result is 6.

Hand the second operand, `k`, to `eval2` for evaluation. `eval1` again waits at this spot while `eval2` runs from the top.

⇒ **Is this form a list?** No

⇒ **Is this form a symbol?** Yes

⇒ **`eval2` returns `k`'s value to `eval1`.** Again, the result is `m`, not 8, because we don't trace through arrows.

`eval1` now has its two arguments, 6 and `m`.

Call the addition function, passing it the arguments, 6 and `m`. The addition function performs the following steps:

◊ **Are there the right number of arguments?** Addition doesn't care how many arguments it is given.

◊ **Are these the right type of arguments?** Addition needs numbers and its second argument is not a number.

◊ **Addition prints a "bad argument type" error on the screen.**

Important Notes

♦ This example is virtually identical to the last one. The only difference is that Example 3 calls the `cons` function, and Example 4 calls the addition function. `cons` doesn't care what kind of arguments it gets, but addition certainly does.

♦ `eval` makes no effort to remedy this situation. It does not evaluate `m` to get 8; if it did it would be tracing through arrows. `eval` doesn't know that this function will blow up. In fact, `eval` doesn't even know what function it is; to `eval` all functions look the same. "bad argument type" is an error message returned by a function, not `eval`.

To ascertain whether an error is detected by `eval` or by the function, a good rule of thumb is to determine whether the `list` function would cause an error with the supplied data. `list` is AutoLISP's most general function because it accepts any type and any number of arguments. Since `list` would be happy with these arguments it must be the function, not `eval`, that complains.

♦ The function does not evaluate `m` to get 8. Only `eval` performs evaluation; a function won't attempt to correct erroneous arguments.

♦ We *must* know the complete contract of each function that we use. It's not enough to know what a function does; we must also be aware of the amount and types of arguments that the function requires.

Here are a few more important points concerning `eval` in general:

- No function other than `eval` evaluates its arguments; all evaluation is done before any function sees its arguments. If the function receives a bad argument, it will issue an error message rather than trying to correct that error.

- `eval` doesn't *type* or *number* check its arguments; `eval` doesn't even know what function it's calling. Once `eval` gets the value of the operator, it has no way of determining what function it has. As our picture notation shows, all functions pretty much look alike. Thus `eval` has no way of knowing what arguments a function requires.

- Occasionally we must call `eval` explicitly, but most often it runs automatically for us. The mechanism that accomplishes this is explained in Section 5.1.4.1.

4.2 The Eval Flowchart Program

The floppy disk enclosed with this book includes a file called `efc.lsp` that contains the Eval Flowchart (EFC) on-line tutorial. When we supply a form to the EFC command, it performs a step-by-step evaluation of that form.

EFC is an extremely valuable aid for learning how different types of forms are evaluated by AutoLISP. Copy this file into your current directory, then load it into AutoLISP as follows:

```
Command: (load "efc")
```

The EFC command can be run by entering

```
Command: efc
Do you want to run EXPERT mode? <Yes>/No: n
Press <enter> when done.
Enter Form:
```

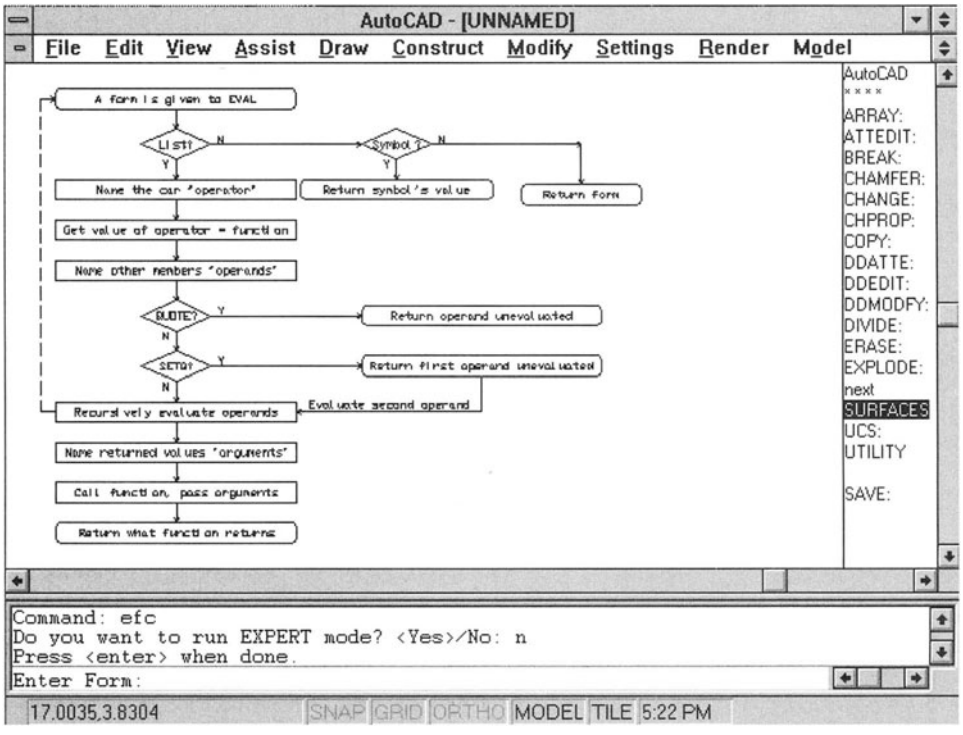


Figure 4-13

The EFC program first asks whether or not you want to run in EXPERT mode. At the start you should probably enter NO (but do try it both ways). EXPERT mode eliminates the pauses so that the flowchart runs faster, but then it's harder to follow what's happening. When the command has completed, flip to the text screen and examine how the code progressed through the flowchart.

The EFC command repeatedly prompts you to enter a form. After you do, hit `<enter>` and the form will be evaluated. It helps to have a printed copy of the real `eval` flowchart for reference. Try some conditions that cause errors and some that require the evaluation of symbols (there are numerous exercises in the labs section).

As the command proceeds, the on-screen "flowchart boxes" change color to highlight the box being executed. Also, when `eval` calls itself, the lines on the left "flash" to indicate another level of recursive function calls. Both of these actions have delays built in so you can see what's happening. If the delay is not long enough (or too long) on your machine, you can modify the program to in-

crease the delay. To do so, alter the integer argument to the `delay` command (in the `recurse` and `switch-color` functions). There is no limit to how high you make this number; you can even set it to 0 to eliminate the delay altogether. If you alter a function, you must reload the file before running EFC again.

To exit the EFC program, simply press `<enter>` at the `Enter Form:` prompt.

The flowchart is set up to run on the basic, standard AutoCAD screen. It erases anything that is already on the screen. The program itself has the following restrictions (some of which will make more sense after you've read the next chapter; they won't affect you now). Any departure from these rules will cause the program to return an error message and possibly exit.

- EFC only works with subrs; it will not handle user-defined functions. `quote` and `setq` are the only special operators it recognizes.
- EFC only type and number checks the subrs `car`, `cdr`, `cadr`, `cons`, `list`, `+`, `-`, `*`, `/`, `quote`, and `setq`. If an expression contains more than one error, EFC might not report the same error as AutoCAD. Also, a single `setq` cannot perform multiple assignments.

4.3 The DRAW Program (Reprise)

In Section 2.7 we were introduced to the DRAW program `(draw.lsp)` for drawing the picture notation of AutoLISP objects on the screen. DRAW has one more capability that we have not yet seen.

If you have the program draw a form such as `(+ 3 4)`, you can thereafter enter the command `run`. DRAW hands the form to `eval` for evaluation and prints (or draws) the result on the graphics screen below the original list. We would run this command as follows:

```
Command: draw
Enter object to draw: (cons 1 2)        ;draw the picture of the form
Command: run                            ;draw the new cons on the screen
```

The following pictures appear on our screen:

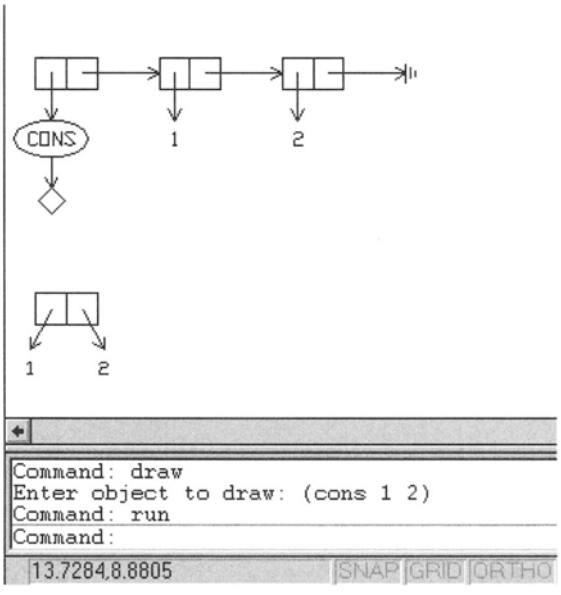


Figure 4-14

4.4 Important Points to Remember

- The argument to `eval` is called a *form.* Any AutoLISP object can be a form.
- If the form is any object except a symbol or a list, `eval` simply returns that form.
- If the form is a symbol, `eval` returns the symbol's value.
- If the form is a cons, that cons must be a list.
- If the form is a list, the car of the list is called the *operator* and the other members are called *operands.*
- An *operator* is a symbol whose value is a function. We always precede an operator with an open parenthesis. Conversely, the first thing past the open parenthesis on any form *must* be an operator.
- An *argument* is an evaluated operand. Arguments are the data that we supply to a function.

- When we write code we use prefix notation. We place the operator first, the operands afterward, and encase the entire expression in parentheses. Ultimately, functions run on arguments. Since the value of a symbol never appears in the printed representation of that symbol, the function is never printed. In many cases, the arguments aren't printed either. Thus, what we see is typically *not* what we get.

- The first line following an error message always shows the exact code where the error occurred.

- It is extremely important that you understand and learn the tasks performed by `eval`. This knowledge will be invaluable throughout your AutoLISP programming career.

4.5 Labs

1. Type the following functions into the EFC program and study how the evaluation process proceeds. Before entering each form, try to predict what will be returned to the screen.

```
Enter Form: (+ 5 6)

Enter Form: (setq a 4)

Enter Form: (setq b a)

Enter Form: (setq c (+ 7 2))

Enter Form: (setq d (+ (a b)))

Enter Form: (setq d (+ a c))

Enter Form: (setq e (+ d (* a 3)))

Enter Form: (/ e a)

Enter Form: e

Enter Form: (/ e (float a))

Enter Form: (setq f (list a c e))

Enter Form: (f e e d)
```

```
Enter Form: (car f)

Enter Form: (car f f)

Enter Form: (cdr f)

Enter Form: (cadr f)

Enter Form: (x y z)
```

2. Enter these forms at the Command: prompt:

```
Command: (setq x 4)
4

Command: (setq y 5)
5
```

For each of the following exercises determine the operator, operands, arguments, and what is returned when we run the function. Figure this out on paper, then test your answers with the EFC program. Keep in mind that if one of the arguments is a list, there will be another level of recursive calls to `eval`. Drawing pictures and using the `eval` flowchart will greatly simplify these exercises.

Here's what we need to write for the expression `(+ 4 (/ 16 2))`:

operator: +
operands: 4 and `(/ 16 2)`
2nd level operator: /
2nd level operands: 16 and 2
2nd level arguments: 16 and 2
2nd level result: 8
arguments: 4 and 8
result: 12

a) `(1+ -7)`
operator:
operand:
argument:
result:

b) `(expt 7 2)`
operator:
operands:
arguments:
result:

c) `(/ x y)`
operator:
operands:
arguments:
result:

d) `(/ (float x) y)`
operator:
operands:
2nd level operator:
2nd level operand:
2nd level argument:
2nd level result:
arguments:
result:

e) `(list x y z)`
operator:
operands:
arguments:
result:

f) `(+ x y z)`
operator:
operands:
arguments:
result:

g) `(cons x y z)`
operator:
operands:
arguments:
result:

h) `(cons x y)`
operator:
operands:
arguments:
result:

i) `(* 4 (+ x (/ 21 3) 2))`
operator:
operands:
2nd level operator:
2nd level operands:
3rd level operator:
3rd level operands:
3rd level arguments:
3rd level result:
2nd level arguments:
2nd level result:
arguments:
result:

j) `x`
result:
(Remember, to learn the value of `x` at the Command: prompt we type `!x`.)

5

Writing and Running Functions

Topics Covered in This Chapter

- The `defun`, `quote`, and `setq` special operators
- The `acad.lsp` file
- The read-eval-print loop
- Global and local variables, system global variables, scoping of variables, and a review of symbols
- Writing and running AutoCAD commands in AutoLISP

Goals for This Chapter

- Gain an understanding of what special operators are and what makes each one special. See the special steps `eval` must take to handle these subrs. Learn the contracts of the basic special operators.
- Learn how to write our own AutoLISP functions. Write functions that not only execute built-in subrs but also call other functions that we have written.
- Learn how to suppress evaluation of operands. Understand why and when we need to do so.
- Discover how the read-eval-print loop affects our coding style. Learn the proper ways to have our functions return results.
- Understand the advantages of local variables. Learn when and how to use them.
- Learn how to write our own AutoCAD commands. See how to incorporate AutoCAD commands into the functions we write.

Introduction

Most of us who have played around with AutoLISP have encountered `defun`, the subr we use to write our own functions. But there's much more to writing functions than just learning how to use this subr.

`defun` is one of several functions that behave in a very peculiar manner. We not only have to learn the contract of `defun`, we must also learn the steps `eval` takes to execute this and other special functions. We'll see how `defun` manages data that we give to it and errors that occur if we don't proceed correctly. We'll also learn how to alter our functions so they become actual AutoCAD commands.

The previous chapter and this one examine at the most basic level the unique ways in which AutoLISP processes code. This is the information we must learn to really understand how AutoLISP works.

5.1 Special Operators

AutoLISP is a very regular language. It has many rules but, once we learn these rules, there are very few exceptions. Alas, there *are* some exceptions to AutoLISP's rules regarding functions; these exceptions are called *special operators*.

Special operators are the "irregular verbs" of the AutoLISP language. They are special because they require `eval` to follow different steps from the ones we've previously studied; they are processed by special code in `eval`.

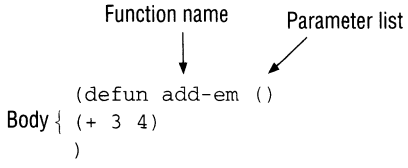
AutoLISP has about a dozen special operators, including `eval` itself. We'll examine three basic ones now: `defun`, `quote`, and `setq`; most of the rest are covered in Sections 6.2 and 6.3. It's critical that you thoroughly understand these three special operators in order to use AutoLISP effectively.

5.1.1 Using `defun` to Define Functions

Until now we have seen how `eval` handles built-in functions (i.e., the subrs). Now we'll learn how to write our own functions, using the special operator `defun`.

`defun` sym list body	Defines a user function or AutoCAD command that performs the tasks specified in its body, and binds the symbol to this function. It returns the name of the function.

`defun`, which stands for "***DE***fine ***FUN***ction," is written as follows:



Each `defun` contains the following components:

`(defun` — We always begin a function definition with `(defun`. When we define a function in an editor, we put the open parenthesis in column 1, then indent the succeeding lines two or three spaces from the left margin in order to make the code more readable. (Code indentation conventions are discussed in Section 13.1.)

Function name — Every function must be given a name. The function name is a *symbol* that is used to access the function when we run it. It can be any valid symbol name, although there are a couple of *dos* and *don'ts* regarding our choice:

DO choose a name that somehow reflects the task that the function performs. For example, the `add-em` function adds numbers together. We could call this function `joe` and it would run just as well. But we wouldn't know what it does when we picked it up six months later.

DON'T choose a name that is one of the built-in subrs. If we call our function `car` or `cons`, we will replace that function definition with our own. This could cause AutoLISP to run erroneously or halt, in which case we'd have to exit our drawing.

Parameter list — In this example the parameter list is empty, but the parentheses are still required. The parameter list is described below.

Body — The *body* is a collection of one or more forms that specify the tasks that the function is being called upon to perform. When the function is run, each form is handed to `eval` in order; thus, the body forms are evaluated sequentially. A function returns whatever the last form evaluated in its body returns.

)	Every function ends with a close parenthesis that matches the open parenthesis preceding the word `defun`.

Two periods of time are associated with functions: a period called *now* when we define a function, and a period called *later* when we run or call the function. We write and load the function into AutoLISP once *now,* then call this function as many times as we want *later.* In other words, we use `defun` to define the function once. Thereafter, we use the function's name to access and run the function as many times as we wish. Whereas we usually *write* functions in the editor, we frequently *run* functions at the Command: prompt.

The built-in subrs work pretty much the same way. For example, the addition function, which was defined by Autodesk's designers when they created AutoLISP, is loaded automatically as part of AutoLISP whenever we enter a drawing. During our drawing session there is no limit to the number of times that we can call it. This function is bound to the symbol +, which we consider to be its name. We can hand this symbol to `eval` whenever we want the addition function to be run.

Once the `add-em` function has been defined, we can call it from the Command: prompt as follows:

```
Command: (add-em)
7
```

If we call it a second time, we get the same result:

```
Command: (add-em)
7
```

Needless to say, the `add-em` function, as it is defined above, is not a very useful function. Every time we call this function it does the exact same thing: it adds the numbers 3 and 4 together.

The reason we create an AutoLISP function is to save time. We specify a set of steps (consisting of AutoCAD commands, subrs, and other functions that we have written) that we would otherwise have to perform by hand and give this collection of steps a name. When we supply that name as an operator to `eval`, it causes the steps to be executed.

If these steps always use the same data, or if we only want to perform them once, then we're better off running them individually in AutoCAD. What makes a function effective is its ability to *execute the same set of steps on different data.*

In its current incarnation, `add-em` is not an effective function. We can increase `add-em`'s usefulness by enabling it to add *any* two numbers together, and by allowing the user to supply these numbers when she calls the function.

This presents a problem: If we allow the user to supply numbers *later* when she calls the function, how can we represent these numbers *now* when we define the function? Furthermore, each time she calls the function she will most likely supply different numbers. *Therefore, when we write the function now, we have no idea what data is actually going to be supplied later when the function is called.*

Since we have no way of knowing what data is going to be supplied to the function, we cannot hard code the data into our routines. Instead of specifying actual data in our `defun`, we use symbols as placeholders for data that will be supplied at a later time. These placeholders are called parameters.

A *parameter* is a symbol that we use in a `defun` in place of data that the user will supply when she calls the function. At that time she provides data via arguments. One of `defun`'s tasks is to bind the parameters to the supplied arguments.

We use parameters all the time in our everyday lives. One example is writing a check. Each check has fields labeled *date, payee, amount,* and *signature.* When the manufacturers printed the checks, they didn't know what we were going to write on each of them. So they put these symbols as indicators or placeholders for the data that we'll supply when we write the check.

To see how data is supplied to a function, let's modify `add-em` so that it uses two parameters and, therefore, accepts two pieces of data when it is called:

```
(defun add-em (n1 n2)
  (+ n1 n2))
```

In this example the parameters, `n1` and `n2`, represent the two numbers that the user must supply when he calls the function. When the `add-em` function is run,

```
Command: (add-em 3 4)
7
```

the parameter `n1` is assigned the value of 3, and the parameter `n2` is assigned the value of 4, providing the following relationships:

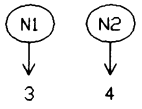


Figure 5-1

The matchup of the parameters and arguments is *positional;* the first parameter is always bound to the first argument, the second parameter is bound to the second argument, and so forth. Once the parameters are assigned values, the forms in the body of the function are sequentially evaluated as previously described. The function returns whatever the last form in its body returns.

`add-em` has only one form in its body, `(+ n1 n2)`. Within this form, + is the operator and `n1` and `n2` are the operands. Evaluation of the operands yields their values, 3 and 4. These arguments are then handed to the addition function, which returns their sum, 7, back to `eval`. This result is printed on the screen.

Let's call the `add-em` function again, this time supplying different data:

```
Command: (add-em 6.5 -2)
4.5
```

Now `n1`'s value is 6.5 and `n2`'s value is –2, but the actions taken are the same: the operands `n1` and `n2` are evaluated, returning the arguments 6.5 and –2, respectively. These are handed to the addition function and the sum, 4.5, is returned to the screen.

As we can see, `add-em` can now add any two numbers. It is a much more general and powerful function than it was originally.

Notes

- The parameter list of a `defun` is never evaluated. `eval` knows that `defun` is a special operator and that its second argument is merely a list of symbols.

- We never give a parameter the same name as the function in which it's defined. In fact, we don't give a parameter the same name as any other symbol used in that function. Remember, a symbol can only have one value, and if we rebind the symbol we lose access to its original value. Fortunately, we're not limited as to our choice of symbol names.

- In its definition of `defun` the *AutoLISP Reference* calls the parameter list an *argument list.* As we now know, parameters are symbols that represent arguments, not the arguments themselves; don't confuse these!

We can define a function that calls other functions, then invoke that function from the Command: prompt in the following manner:

```
(defun add-twice (num1 num2)
   (+ (add-em num1 num2)
      (add-em num1 num2)))

Command: (add-twice 8 3)
22
```

The `add-twice` function has two parameters, `num1` and `num2`; thus we supply two arguments, 8 and 3, when we call it. `add-twice` calls the `add-em` function two times, and the `+` function as well. What happens when `add-twice` calls `add-em`?

The form `(add-em num1 num2)` is treated like any other form we saw in the `eval` chapter. The symbol `add-em` is the operator. `eval` finds the value of the operator, which is the `add-em` function. It then evaluates the two operands, 8 and 3. The 8 and 3 are returned as arguments and passed on to the `add-em` function, which adds them together and returns 11. This operation is performed twice and the results are added, yielding the final result, 22.

The important concept to understand here is that we can have as many levels of nested function calls as are needed to accomplish a particular task. `eval` repeatedly calls itself until it encounters the lowest nesting level, which ensures that each nested function is executed.

Although Autodesk has named the built-in functions *subrs,* there is no logical difference between these and the functions that we write; `eval` treats both types the same. Therefore, when we write a function, we actually add functionality to AutoLISP . . . we expand the language. Our functions can call other functions that we have written as readily as they can invoke the subrs.

There *are* minor differences between our functions and subrs. For example, their printed representations are unalike. We've seen that built-in AutoLISP functions have an uninformative printed representation:

```
Command: !cons
<subr: #87f0169e>
```

When we print one of our own functions, we see the code it contains:

```
Command: !add-twice
((NUM1 NUM2) (+ (ADD-EM NUM1 NUM2) (ADD-EM NUM1 NUM2)))
```

Note that the printed representation of the function does not include its name. This is because the function's "name" is the symbol that is bound to the function and not really part of the function itself.

When AutoLISP encounters an error, it prints the error message followed by the code where the error occurred. As we shall see, our ability to read the printed representation of our code greatly simplifies the debugging process.

♦ ♦ ♦

Let's examine the actions taken by `defun` when we define the `add-twice` function at the Command: prompt or load it from a file:

1. `defun` creates (in the AutoLISP world) a function that takes two numbers as arguments, adds them together twice, and returns their sum.

Figure 5-2

2. It then makes this function be the value of the symbol `add-twice`.

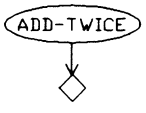


Figure 5-3

It is important to recognize that `defun` does not *run* the function, it merely *defines* it. To run it we might enter the following:

```
Command: (add-twice 3 4)
14
```

Although we have the choice of writing our code using an editor or directly at the Command: prompt, we nearly always write `defuns` using an editor. One reason is that if we make a mistake at the Command: prompt, we must retype our entire code; AutoCAD does not have a "DOSKEY" feature to bring back previously entered lines of text.

A more important reason for using an editor goes back to the heart of why we write programs in the first place: to be able to quickly repeat a series of steps. If, for example, we want to put a single door into a door frame, it's certainly easier to do it in AutoCAD than it is to write a program to accomplish the task.

The power of programming is realized when we need to put many doors into many door frames. When we write a program to perform such a task, we usually intend to run the program on numerous occasions. Unfortunately, everything we type at the Command: prompt is lost when we exit AutoCAD. By using an editor to write our programs to a file, we can load that program whenever we need to run it and give the file to our coworkers as well.

When we write a function using an editor, the function must be loaded into the AutoLISP world before it can be run. As far as AutoLISP is concerned, this process is identical to entering a function at the Command: prompt; `eval` evaluates the function as it is being loaded. When we load a file containing a single `defun`, we must remember to specify the name of the file, not the name of the function.

When we load a file containing several `defuns`, each function is defined and assigned to the appropriate symbol (i.e., the function name), making it ready to run. AutoLISP doesn't care what order we place our functions in a file; when the file is loaded, all of the functions are defined. Thereafter, any function can call any other defined function.

As we now know, we must supply the same number of arguments as there are parameters. Otherwise we will get an error message:

```
Command: (add-twice 3 4 5)
error: incorrect number of arguments to a function
```

We must also supply the right type of arguments or we get a "bad argument type" error. However, unlike subrs, our own functions do not automatically perform type checking on their arguments. We can add type checking into our code, but typically we don't. To learn why this is true, examine the following function call:

```
Command: (add-em 3 nil)
error: bad argument type
(+ N1 N2)
(ADD-EM 3 nil)
```

When the addition function is called, it issues an error message because the supplied argument, `nil`, is not a number. Given that the subrs test for these errors, we don't need to have our own functions incur the overhead of checking for incorrect data themselves. On the other hand, we often do want our AutoCAD *commands* to check for errors so they don't crash when our drafters run them.

Parameter passing is a difficult concept to understand the first time it is encountered, and it may take some practice before you really understand how it works. Let's look at another way we can pass data to the `add-em` function:

```
Command: (setq x 3)
3

Command: (setq y 4)
4

Command: (add-em x y)
7
```

In this example our operands are symbols, but `add-em` still returns the correct result because the *arguments* are numbers. It's worth examining this example closely so that we really understand what `eval` is doing and to verify that `eval` handles functions we write in the same way it handles subrs. You'll probably find it clearer if you follow along with the `eval` flowchart.

First, the form is handed to `eval` Level 1. This form is the list `(add-em x y)`.

Name the car of the list the "operator." The symbol `add-em` is the operator.

Get the value of the operator and save it for later on. This is the `add-em` function.

Recursively evaluate the operands. The operands are `x` and `y`.

⇒ **Evaluation of the operands yields the arguments.** The arguments are 3 and 4.

Call the `add-em` function, passing it the arguments. The parameters, `n1` and `n2` are bound to 3 and 4, respectively.

◇ **`add-em` sequentially evaluates the forms in its body.** The list `(+ n1 n2)` is the only form in `add-em`'s body, so it is evaluated next.

◇ `+` is the operator, `n1` and `n2` are the operands. The operands are evaluated and the arguments, 3 and 4, are handed to the addition function.

◇ **The addition function verifies it has the right type of arguments, then adds them.**

The result 7 is handed back through `eval` to the `add-em` function. Since this is the last form in the body of the `defun`, `add-em` returns 7 as well.

Observe how the operands and parameters work together. Initially, `eval` finds the value of the operand `x`, which is 3. That number is handed to the `add-em` function *and made the value of the parameter,* `n1`. When the body of the function is subsequently evaluated, the parameter `n1` is handed to `eval` and 3 is returned again.

Notice that there is a double evaluation. `x` and `y` are evaluated to get the arguments, 3 and 4, respectively. These arguments are passed to the `add-em` function, where they are bound to `n1` and `n2`. The act of running a function consists of evaluating the forms in the function's body. When `n1` and `n2` are evaluated, the numbers 3 and 4 are returned again.

In this example, two different symbols are used to represent the argument 3. Outside the function we refer to it by the symbol `x`; within the function the symbol `n1` is used to access this number. It doesn't matter what we call the number because `eval` is going to strip off the symbol anyway. The symbol is just a name; it's the number that we're after.

Here's an analogy: When I talk to my brother I call him "Joel." However, I want to speak to the person, not his name. "Joel" is just a symbol that our parents "`setq`'d" him to, a way of identifying that person. Likewise, when others address him as "Mr. Harkow" or "Dr. Harkow" (he's a physician), they are referring to the same individual. The titles are just means of accessing the person.

We use symbols to name data in a similar manner. `n1` and `x` just furnish the means for accessing a variable piece of data. It doesn't matter what we call them because `eval` is going to go past the symbols each time and return their values, which is the data that we really want.

Using parameters to represent our data provides flexibility because we can use any symbols we want to name the data. When we call a function, we don't have to know what symbols it uses to access our data.

♦ ♦ ♦

Let's look at some examples that require us to interpret a problem statement and create a `defun` that solves the problem. In each case we must begin by determining the contract of the function: how many and what kind of arguments the function takes, what the function does, and the result that it returns.

See if you can solve each problem before looking at the solutions that follow. Then test the function in AutoLISP to verify your results.

Example 1

AutoLISP generally uses *radians* to measure angles. But since radians are hard for people to work with, we usually measure angles in degrees. What we'd really like to do is work in degrees but supply radians to functions that require them. Let's write a function called `dtr` to accomplish this.

A circle is divided into `2 * pi` radians, meaning there are `pi` radians in 180 degrees. Thus, the formula for converting degrees to radians is `r = pi * degrees / 180.0`. Once we have written the `dtr` function, we would invoke it in the following manner:

```
Command: (dtr 180)
3.14159

Command: (dtr 90)
1.5708
```

Our first step in writing this function is to determine the arguments that the function is to take. Since its task is to convert degrees to radians, we can deduce that it must be supplied with a number representing degrees as its only argument.

Since the argument is the number of degrees, a logical name for the parameter that represents it is `deg`. We could call it `num` because it represents a number, but `deg` is a little more informative. We can now write the first line of our function:

```
(defun dtr (deg)
```

The body of the function consists of the expression that was supplied to us in the statement of the problem. It must, however, be converted to prefix notation. The final `dtr` function is as follows:

```
(defun dtr (deg)
  (/ (* pi deg) 180.0))
```

The `dtr` function can also be written in the following way:

```
(defun dtr (deg)
  (* pi (/ deg 180.0)))
```

These two functions work identically; we get the same result regardless of the order in which the arithmetic is performed. Most of the time, however, we must execute tasks in a specific order.

Keep in mind that the result of an integer division is an integer. Suppose we wrote the second version of `dtr` using 180 instead of 180.0:

```
(defun dtr (deg)
  (* pi (/ deg 180)))
```

This will give the wrong answer if we supply an integer argument. For example, if the argument is the number 90, the function will calculate `(/ 90 180)`, which is 0, and return 0 as the result. We must make at least one of the arguments to division be a real number in order to force a real division to be performed. Using 180.0 satisfies this requirement.

Note that we *can* use 180 in the first version of `dtr` because the multiplication is done before the division. Since the value of `pi` is a real number, the result of the multiplication is real, and a real division is performed.

Given the original problem statement, we might be inclined to write the `dtr` function as follows:

```
(defun dtr (deg)
  (setq r (/ (* pi deg) 180.0)))
```

Although the use of `setq` in this instance isn't wrong, it is not preferred. We almost never use `setq` on the last line of a `defun`. The reasons for this are explained in Section 13.1.1.

Note: We invoked this function by typing `(dtr 180)` at the Command: prompt. This is an example of how we might call `dtr` once it has been written. It has nothing to do with the actual writing of the `dtr` function, and the argument, 180, is not part of the `dtr` function itself.

Example 2

Write a function that takes a list containing two elements and returns a new list with the elements interchanged. Run the function in the following manner:

```
Command: (swap (list 2 3))
(3 2)
```

Figure 5-4 depicts the initial list and the resultant list:

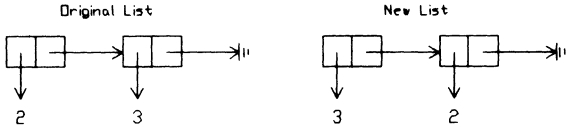


Figure 5-4

Before going further, read the problem statement again and ask yourself the following question: "How many and what kind of arguments does this function take?"

If you said it takes two arguments, then read the problem statement more closely. It specifies that the sole argument to the function is *a list.* The fact that the list will contain two elements is irrelevant. Even if the function requested a list of 100 elements, it would still expect a single list as its argument.

Because the function takes a single argument, we must provide the function with one parameter to match it. Because the argument is a list, let's choose the symbol `lst` to represent it. We may be tempted to name the parameter `list`, but if we do so we will lose access to the `list` subr. The first line of the function is written as shown:

```
(defun swap (lst)
```

Since `lst` is the parameter, within the function it is bound to the supplied list:

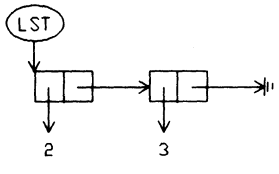


Figure 5-5

Now we can use the symbol `lst` to access the backbone conses and elements in this list.

Having decided upon the function's name and parameter, we must next write the body. This function's task is to create a new list with the elements of the original list interchanged. The problem requires that we create a *new list,* which suggests to us that we use the `list` function (it's a good thing we didn't lose access to it by calling our parameter `list`!):

```
(defun swap (lst)
  (list
```

What is the first element of our new list going to be? Can we express it in terms of the parameter, `lst`?

We want the first element of our new list to be 3, which is the second element of the original list. We can access it by entering `(cadr lst)`. Similarly,

`(car lst)` will return the first element of our original list. Thus we can make our `swap` function create the new list in the following manner:

```
(defun swap (lst)
  (list (cadr lst) (car lst)))
```

Now that we have written the function once, we can call it many times and have it operate on different data each time:

```
Command: (swap (list "abc" 6.4))
(6.4 "abc")

Command: (setq m (list 2 (list 3 (list 4))))
(2 (3 (4)))

Command: (swap m)
((3 (4)) 2)
```

The first example shows that `swap` works with any two-element list, not just numbers.

In the second example we make our original list, `(2 (3 (4)))`, be the value of the variable `m`. As Figure 5-6 shows, this is a two-element list consisting of the number 2 and the list `(3 (4))`:

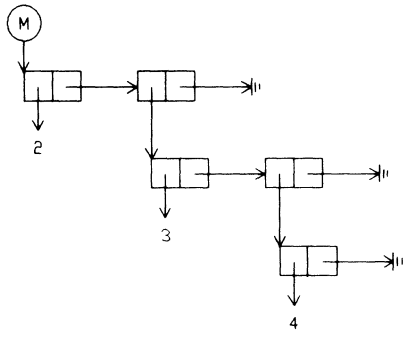


Figure 5-6

When we type the form `(swap m)` at the Command: prompt, `swap` is the operator and `m` is the operand. The operand is evaluated and its value, the list `(2 (3 (4)))`, is the argument that is handed to the `swap` function. Within the `swap` function, this argument is bound to the parameter, `lst`, and the function proceeds as before.

Once again, notice the subtle use of symbols in this example. Outside the function we access the list through the symbol `m`. Within the function the symbol

`lst` is used to locate the *exact same list.* We're free to name this list anything we want (or, as in the first examples, leave the list nameless), because the function accesses any list we supply via the symbol `lst`.

Note: `swap` uses the `list` function to build a list. However, not all functions that *use* lists require that we *create* a list. Only use the `list` function when you intend to build a list.

Example 3

In Chapter 3 we learned the contracts for the `sin` and `cos` functions and discovered that AutoLISP does not possess a tangent function. Write a function called `tan` that computes the tangent of a supplied angle by dividing the sine of the angle by the cosine of the angle.

Since the solution requires the use of `sin` and `cos`, our first step might be to go back to Chapter 3 and review the contracts of these functions. When we do, we learn that the angle must be supplied in radians. Thus we have an immediate use for the `dtr` function that we wrote in Example 1.

Once written, we'd run the `tan` function as follows:

```
Command: (tan 45)
1.0
```

Since the function's job is to determine the tangent of an angle, we'd like to supply it with one argument: the angle expressed in degrees. What's a good name for the parameter? `deg` would work, but let's try `angle` for variety. We can now write the first line of our function:

```
(defun tan (angle)
```

The body of the function must divide the sine of `angle` by the cosine of `angle`. However, prior to the division we must convert the angle from degrees to radians:

```
(defun tan (angle)
  (setq angle (dtr angle))
```

The second line of code finds the value of the symbol `angle`, converts it to radians, then rebinds `angle` to this new value. Now we can perform the division:

```
(defun tan (angle)
  (setq angle (dtr angle))
  (/ (sin angle) (cos angle)))
```

Be aware that in this example `cos` could return 0, which would cause a "divide by zero" error.

Notes

- We don't `setq angle` on the last line. Since `defun` returns the result, a `setq` would be superfluous.

- Any function can call any other function. The `tan` function invokes the `dtr` function just as it would any subr.

- Some programmers prefer to put the final close parenthesis on its own line, under the matching open parenthesis:

```
(defun tan (angle)
  (setq angle (dtr angle))
  (/ (sin angle) (cos angle))
)       ;end defun
```

  When doing so, we might also add a comment that indicates what structure is being closed. Doing this at the end of each major construct within a `defun` can make a large piece of code more readable.

- There is another way we could write the `tan` function:

```
(defun tan (angle)
  (/ (sin (dtr angle)) (cos (dtr angle)))
)         ;end defun
```

  Here we save a `setq` at the cost of running the `dtr` function twice. Which way is better? If `dtr` was a lengthy function we'd probably only want to run it once. On the other hand, if we'd like to squeeze many functions on the screen, we might begrudge the extra line that the `setq` uses. There are often several equally effective ways to accomplish a task in AutoLISP. Choose the algorithm that makes the most sense to you.

Subrs such as addition and multiplication allow us to supply as many arguments as we wish, but AutoLISP doesn't let us provide that capability for our own functions. We can overcome this restriction by having the user supply the necessary data in a list, then using one of the looping functions to access the elements. For example, we could write a function to add the members of a list:

```
(defun add-list (lst)
    .
    .
    .)
```

When she calls the add-list function, the user can supply as many numbers as she would like in a list:

```
Command: (add-list (list 2 6.5 7 -3 9))
21.5
```

Another way we can make our functions accept unlimited input is by writing a loop that repeatedly prompts the user to enter data until some terminating condition is met. This is analogous to the way the LINE command requests its points.

5.1.1.1 The acad.lsp File

If we want certain files to be loaded every time we enter AutoCAD, we place them in a file called acad.lsp. The acad.lsp file is analogous to the acad.dwg file. Whereas acad.dwg contains information that initializes our drawing environment, acad.lsp helps initialize our AutoLISP environment.

Whenever we enter AutoCAD or open a new drawing, AutoCAD checks to see whether we have an acad.lsp file and, if so, loads it. In this file we put commonly used utilities that we always want present whenever we begin a drawing session.

The examples we've just seen contain two candidates for the acad.lsp file: dtr and tan. If we frequently work with angles, we might want these functions to always be present. Having them in our acad.lsp file eliminates the need to find and load them when they're needed.

How do we decide whether a function belongs in acad.lsp? If it's something we use often, it should probably be included. If we use it once a week or less, we might not want it present. If our acad.lsp file gets too large, it takes longer to load, eats up valuable memory, and becomes more difficult to manage.

The acad.lsp file usually consists mostly of defuns but can also contain any AutoLISP function we'd like to have run as a script whenever we open a drawing. For example, suppose we're working on a project for the highway department that requires the use of certain AutoLISP functions. We'll put these functions in a file called hwy.lsp, then add the line (load "hwy") to our acad.lsp file. Then, whenever we enter AutoCAD, hwy.lsp will be automatically loaded. When the project is finished, we simply remove this line from the acad.lsp file. This saves us the hassle of adding a whole bunch of functions to acad.lsp then having to remove them later.

5.1.2 Using quote to Prevent Evaluation

quote is a special operator whose job it is to inhibit evaluation. Of all of the functions in AutoLISP, quote is the function that gives new users the most trouble. The correct use of quote will probably eliminate 25 percent of our errors.

Think about this question for a minute: How can we create a cons whose car is the symbol a and whose cdr is the symbol b?

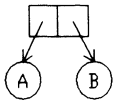


Figure 5-7

At first blush you might suggest entering the following form:

```
Command: (cons a b)
```

But what happens when we do so?

When eval is given this form, it evaluates the two operands, a and b. Since they have not been explicitly assigned values, both symbols have a value of nil. Thus, the cons function builds a cons whose car is nil and whose cdr is nil. This is a list whose only element is nil:

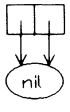


Figure 5-8

Obviously, this is not what we want. As surprising as it might seem, we haven't yet seen how to build the cons (A . B). To create such a cons we must tell eval, "Don't evaluate the symbols a and b; just return them as is." We can accomplish this task with the special operator quote.

quote obj	Takes an AutoLISP object as an operand and returns that operand unevaluated.

quote enables us to build the cons we want:

```
Command: (cons (quote a) (quote b))
(A . B)
```

Here's the structure AutoLISP creates when we enter this form:

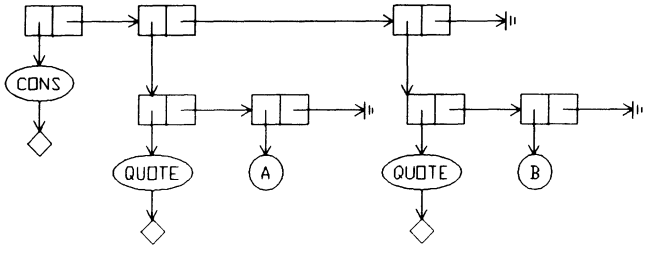


Figure 5-9

Why must quote be a special operator?

To see why, let's assume for a moment that quote is not a special operator and that the form (quote a) goes through the normal eval process. Then eval performs the following steps:

The form is a list, so name the car the "operator." The symbol quote is the operator.

Get its value and save it for later on. This is the quote function.

Name the other member of the list the "operand." The symbol a is the operand.

Recursively evaluate the operand. Hand the operand, a, to eval Level 2 for evaluation.

⇒ **Since a is a symbol, eval Level 2 returns its value, nil, to eval Level 1.**

Call the quote function, passing it the argument, nil.

Oops, we're now stuck! Since we cannot backtrace arrows, the quote function has no way of accessing the symbol a. In other words, when we try to go through the normal eval process for quote, the function receives its argument too late.

To avoid this problem, eval must test whether the operator is quote *before* sending the operand to the next eval level for evaluation. If it is indeed quote,

eval skips the evaluation process for that operand. Thus, for (quote a), eval actually undertakes the following actions:

The form is a list, so name the car the "operator". The symbol quote is the operator.

Get its value and save it for later on. This is the quote function.

Name the other member of the list the "operand." The symbol a is the operand.

Is the operator quote? Yes.

Call the quote function, passing it the operand, a.

◇ **quote returns the operand to eval.**

eval returns the result. The result is a.

quote is a special operator because it requires eval to do something different from its normal task. For quote, eval must run special code that inhibits evaluation of the operand. Most special operators are special because they have one or more operands that are "quoted," that is, not evaluated. But quote is the only special operator whose sole purpose is to inhibit evaluation. As part of its "specialness," quote is given an *operand,* not an *argument.*

5.1.2.1 Using quote with Symbols

Let's look at several applications for quote, beginning with symbols. Type these examples at the Command: prompt to see how they run. Initialize the environment by assigning a value to the variable x, then create a couple of conses, as follows:

```
Command: (setq x 4)
4

Command: (cons 2 x))
(2 . 4)

Command: (cons 2 (quote x))
(2 . X)
```

The first cons function builds a cons in a manner with which we're familiar by now. The operand, x, is evaluated, yielding the argument, 4. A cons is built with 2 and 4 as its car and cdr.

In the second example we quote x. x is not evaluated and is handed to the cons function as the second argument. The function builds and returns a cons with a car of 2 and a cdr of x.

Here is another pair of examples that show the effects of quote:

```
Command: (+ 2 x)
6

Command: (+ 2 (quote x))
error: bad argument type
```

In the first example the addition function is given the arguments 2 and 4 (the value of x), so it returns 6.

The second example is more problematic. Once again we ask eval to not evaluate x, and it happily complies. But in this case the symbol x itself is handed to the addition function, which is a no-no. Since addition requires numbers as arguments, it is forced to issue a "bad argument type" error.

5.1.2.2 Using quote with Lists

The creation of lists frequently involves the use of quote; there are three methods that are most commonly tried. In each of these examples the pictures show the code that is handed to eval for evaluation.

Example 1

```
Command: (setq l (list 6 7 8))
(6 7 8)
```

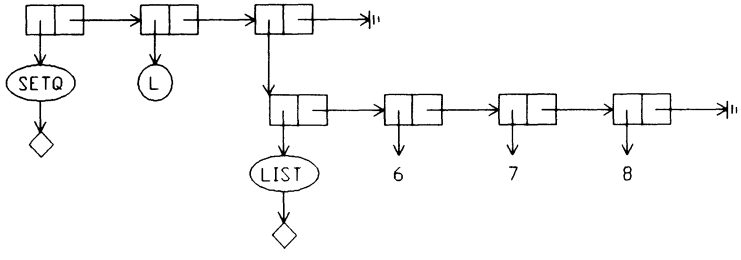


Figure 5-10

In the first example we provide the list function with three numbers as arguments, and it builds and returns the new list.

Example 2

```
Command: (setq l (quote (6 7 8)))
(6 7 8)
```

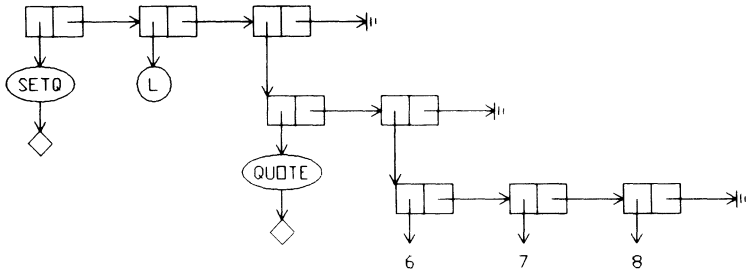


Figure 5-11

The next example shows a different way to achieve the same result. The second argument to `setq` is a list whose car is the symbol `quote`. The form `(quote (6 7 8))` has one operand, the list `(6 7 8)`. Remember that `quote` returns this operand unevaluated. Thus, the list is returned as the second argument to `setq` and is made the value of `l`.

The difference is that in the first example we have the `list` function create a list for us. In the second example we tell `eval`, "Here is the list `(6 7 8)`. Don't evaluate it; it's data, not code. Just assign it to the variable `l`."

Example 3

```
Command: (setq l (6 7 8))
error: bad function
```

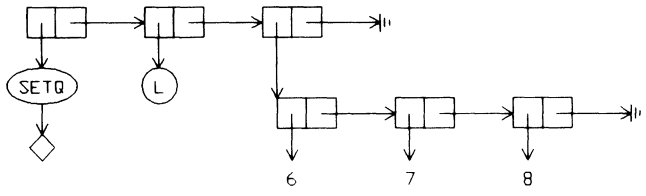


Figure 5-12

In the last example our second operand is a list that is not quoted. Do you see why it returns a "bad function" error? When `eval` Level 2 receives the list `(6 7 8)`, it names the car of the list, 6, the "operator" and tries to obtain its value. Since 6 is not a valid operator, `eval` issues the error message.

5.1.2.3 Using quote with Other Object Types

Let's see what happens when we use quote with numbers:

```
Command: (+ 5 2)
7

Command: (+ (quote 5) (quote 2))
7
```

As we can readily see, the results are the same for both examples. Why?

Look again at the eval flowchart. When eval is handed a number, it simply returns the number. Thus, it doesn't matter whether or not we quote numbers, the results returned are always the same.

Numbers are called *self-evaluating forms,* because eval always returns any number it is given. The same is true of strings. In fact, as the flowchart shows, all datatypes except symbols and lists are self-evaluating. This means that we *never* have to quote them.

Is it wrong to quote a number or a string? No, but if we do, someone reading our code will say, "That person quoted a number; I guess he really doesn't understand how AutoLISP works." Therefore, even though eval doesn't care whether these objects are quoted, it is frowned upon in a stylistic sense.

5.1.2.4 The Quote Character: '

The quote special operator is used so frequently, we have an abbreviation for it. Instead of typing (quote x), we can simply enter 'x. We replace the parentheses and the symbol quote with a ', which is the keyboard quote character. This ' has nothing whatsoever to do with the ' we use to indicate a transparent AutoCAD command.

Naturally, 'x is easier to type than (quote x) and is therefore preferable to the longer version. In all probability, we will never have a need or a desire to type in the long form. However, it is important to know the longer version and to be aware that the two forms are expressing the exact same construct.

When we enter code at the Command: prompt, a reader picks up what we type and builds it internally as AutoLISP objects. (The AutoCAD reader is discussed in Section 5.1.4.) This reader recognizes the quote character and converts 'x to (quote x). Only the reader knows the quote character; the rest of AutoLISP just sees the longer version.

Whenever AutoLISP prints an error message, it includes the code where the error occurs. Although the offending code may contain a quote character, the resultant error message *always* shows the longer form of `quote`:

```
Command: (+ 'x 3)
error: bad argument type
(+ (QUOTE X) 3)
```

Again, this is because the reader interprets the quote character and converts it to the longer form. To fully understand the error messages we receive, we must be aware that `'` is just an abbreviation for the actual AutoLISP `quote` function. When writing code, however, we always use the shorter version.

The quote character can replace the `quote` function wherever we need to suppress evaluation. Thus,

```
Command: (setq l (quote (a b c)))
(A B C)
```

is equivalent to

```
Command: (setq l '(a b c))
(A B C)
```

Similarly,

```
Command: (setq m (cons (quote a) (quote b)))
(A . B)
```

is equivalent to

```
Command: (setq m '(a . b))
(A . B)
```

Notes

- When we enter the expression `'a`, `eval` is handed the form `(quote a)`. `quote` is a function name and, as such, a valid operator.
- We never call the quote character an "apostrophe"; we always call it "quote" because that's the name of the actual AutoLISP subr. The quotes we place on either side of a string (as in `"test.lsp"`) are called "double quotes."

5.1.2.5 The Difference between Data and Code in AutoLISP

In most computer languages, data and code have markedly different appearances. One unusual aspect of AutoLISP is that there is very little differentiation between the two. For example,

```
(+ 3 4)
```

is usually code, but

```
(2 3 4)
```

is data.

Whether something is code or data depends on how it is used. AutoLISP assumes that everything is data, until it is given as a form to `eval` for evaluation. At that moment it becomes code.

For example, `(+ 3 4)` is simply a list that happens to have the symbol + as its car. To AutoLISP, it is no different from the list `(2 3 4)` *until it is evaluated.* Then + becomes an operator and `(+ 3 4)` is assumed to be executable code.

`(2 3 4)` is treated the same way by AutoLISP: It too is data until it is handed as a form to `eval` for evaluation. However, `(2 3 4)` can never be code because 2 is not a valid operator. If we attempt to have it evaluated, `eval` will return a "bad function" error. Therefore, to prevent evaluation of this list, we must quote it before handing it to `eval`.

`quote` is an extremely important special operator that is probably the most misused function in the entire language. New users tend to ascribe all sorts of weird properties to `quote`, yet its contract is really very simple: `quote` *inhibits evaluation of its single operand.* Period.

From now on, whenever we type in a line of code at the Command: prompt or in a program, we must examine each operand and ask ourselves whether we want that operand evaluated. If so, we can't quote it; if not, we must quote it. *The decision whether or not to quote an operand is never arbitrary.*

5.1.2.6 Guidelines for Using `quote`

Initially, we might be confused about whether or not to quote an operand. Here are some guidelines:

- Conses (dotted pairs) must always be quoted; they're never code.
- All other objects, except symbols and lists, should never be quoted; they evaluate to themselves.

- When we want to supply a symbol as an argument to a function, we must quote that symbol. If we don't quote it, then the symbol's *value* will be handed to the function. Given that x's value is 4, this example shows what happens when we do and don't use `quote`:

```
Command: (cons 'x x)
(X . 4)
```

- Never quote `defun`'s parameter list. `defun` is a special operator whose parameter list is never evaluated.

  Don't quote parameters in the parameter list or in the body of a `defun`. Since parameters represent the data that is supplied to a function, we virtually always want them evaluated.

- Whether or not we quote a list depends on how we intend to use it. If the list is data, then we *don't* want it evaluated, so we must quote it:

```
Command: (setq l '(a b c))
(A B C)
```

  If the list is code, then we *do* want it evaluated, and therefore don't quote it:

```
Command: (setq m (+ 2 3))
5
```

5.1.3 A Closer Look at `setq`

`setq` is a special operator we have seen before. Given our new knowledge, it is worth examining its contract a little more closely. `setq` takes a symbol and any AutoLISP object and binds the symbol to that object. If no variable exists with that name, a new one is created.

`setq` is a special operator for much the same reasons that `quote` is; namely, that its first operand is not evaluated. `eval` must specially test for `setq` to ensure that the evaluation is not performed. On the other hand, the second operand is evaluated in the normal manner.

One hitherto unseen facet of `setq` is that we can use it to set the values of several variables in one statement:

```
(setq a  5
      a2 6
      b (+ a a2))
```

In this example, a is given the value 5 and a2 is assigned the value 6. The value of b becomes the sum of a plus a2, 11. This, then, is the complete contract for setq:

setq sym obj . . . sym obj	Takes pairs of operands. The first of each pair is a symbol that isn't evaluated. The second of each pair is evaluated and made the value of its partner symbol. It returns whatever the last form evaluated returns.

We always provide setq with one or more *pairs* of operands. The first member of each pair *must* be a symbol; it's not evaluated. The second member of the pair *is* evaluated, and the result is assigned to the first member of that pair.

Stylistically, when we write AutoLISP code in a file, we usually align each pair of arguments vertically, as shown above. This makes it easier to differentiate each symbol from its value. As far as AutoLISP is concerned we could have put all of our assignments on a single line. Indeed, when we perform multiple assignments at the Command: prompt, we typically do write the entire expression on one line, like so:

```
Command: (setq a 5 a2 6 b (+ a a2))
11
```

When the second member of a pair is an involved form, this format becomes very difficult to read. However, when we setq several variables at the Command: prompt, we usually don't care about the format.

We must be careful not to setq any variable that is also the name of a function or we'll lose the function definition. For example, if we enter (setq list '(a b)), then we no longer have access to the list function. Future attempts to use it will result in a "bad function" error. If this occurs we simply exit, then reenter our drawing . . . AutoLISP is automatically reloaded.

5.1.4 How Code Is Executed at the Command: Prompt

How does AutoCAD know whether what we type at the Command: prompt is an AutoCAD command or AutoLISP code? How does it process what we enter? If we know what actions are taken at the Command: prompt, we can present our data and code correctly and have it executed properly.

A *reader* exists whose job it is to interpret what we enter at the Command: prompt. Once the Command: prompt has been issued, the reader stops all processing and

waits for us to enter some data, terminated by a <space> or <enter>. When we have finished our input, the reader performs the following steps:

1. It first checks whether we entered a symbol. If so, it asks whether that symbol is an AutoCAD command, such as ZOOM. If it is, the reader has AutoCAD run that command.

2. If the symbol is not a valid AutoCAD command, the reader checks to see whether it is the name of a DOS command, such as TYPE or SHELL, that has been defined in the `acad.pgp` file. If so, it runs that command. If not, the reader issues the message:

   ```
   Unknown command. Type? for list of commands.
   ```

   Suppose we assign a value of 5 to `y`, then try to learn its value at the Command: prompt:

   ```
   Command: (setq y 5)
   5

   Command: y
   Unknown command. Type ? for list of commands.
   ```

   Here the reader thinks we're trying to run an AutoCAD command. When we use symbols inside of our `defuns`, AutoLISP knows we want the symbols evaluated. But when typing to the reader at the Command: prompt, we must have a way of differentiating between AutoCAD commands we want executed and AutoLISP symbols we want evaluated. This is why, when we want to learn the value of a symbol at the Command: prompt, we must precede it with an exclamation point:

   ```
   Command: !y
   5
   ```

   The exclamation point tells the reader that what follows is an AutoLISP form, not an AutoCAD command.

3. When we enter either a symbol preceded by an exclamation point or an expression in parentheses, such as `(+ 3 4)`, the reader knows that what we typed is an AutoLISP form that we want evaluated. Rather than calling AutoCAD, it runs a process known as the read-eval-print loop.

5.1.4.1 The Read-Eval-Print Loop

The *read-eval-print loop* is a process that runs whenever the reader notices that we have entered an AutoLISP form at the Command: prompt. The form may be a list or it might be a symbol that follows an exclamation point. The loop entails the following steps:

1. The reader picks up what we have typed and builds the appropriate cons structure. For example, if we enter

   ```
   Command: (setq x (+ 3 4))
   ```

   the reader builds the following list:

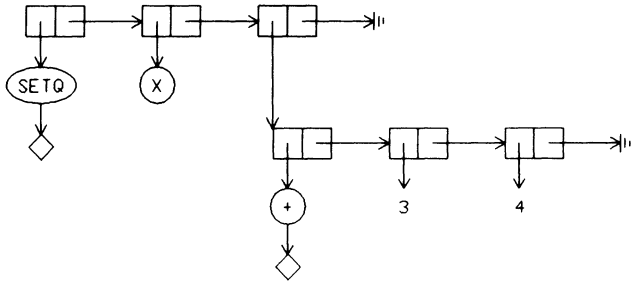
   

   Figure 5-13

2. The reader then hands this structure to `eval`, which performs the evaluation and returns its result, 7, to the `print` function.

3. `print` displays the 7 on our screen and reissues the Command: prompt. The loop then repeats.

If we type `!x` at the Command: prompt, the same thing happens. `x` is evaluated and its value is returned to `print`, which prints it on the screen. Because the read-eval-print loop continually runs at the Command: prompt, we rarely need to explicitly call `eval`.

Note, incidentally, that this is the same reader that determines that `'a` is not a transparent command. It converts `'a` to `(quote a)` and builds the list internally.

The read-eval-print loop allows for interactive input at the Command: prompt and is what makes AutoLISP an interpretive language. It is also the cause of a very peculiar occurrence. Having assigned a value of 7 to `x`, suppose we type `(print x)` at the Command: prompt. What do you suppose will happen?

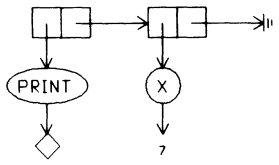


Figure 5-14

As it did before, the reader builds the AutoLISP form we entered and sends it to be evaluated. `eval` obtains the value of `x`, then calls the `print` function with this value. `print` prints 7 on the screen, then returns it back to `eval`. `eval` then returns that value to the `print` function as part of the read-eval-print loop, and *the 7 is printed a second time!*

```
Command: (print x)
7 7
```

The result is printed once because we requested it by running the `print` function; it is printed a second time because we ran this function at the Command: prompt. When we place a `print` function in the middle of a `defun`, its argument is printed only once. In this example it's printed a second time specifically because it went through the read-eval-print loop.

The read-eval-print loop has ramifications for the way we write functions. Let's write a function that takes one parameter, `m`. Perhaps this function performs a number of calculations, then prints the final value of `m` when it is done:

```
(defun change-m (m)
    .
    .
    .
  (print m))
```

Suppose we run this function and the final value of `m` is 8. Then a function call at the Command: prompt would appear as follows:

```
Command: (change-m 2)
8 8
```

Recall that a function returns whatever the last form in its body returns. Therefore, this function not only prints the value of `m`, it returns it as well. When we run the function at the Command: prompt, the result is printed a second time. Having our results printed twice is not the worst calamity that could befall our AutoLISP code, but it isn't pretty to look at. So how can we avoid this occurrence?

Because the problem is a direct result of having the last line of our `defun` be a `print` function, the solution is to *never make* `print` *be the last line of a function!* `print` has a number of variants, such as `princ`, `prin1`, `prompt`, and `write-line`. We typically don't put any of these on the last line of a function either.

This gives rise to the obvious question, "If we don't want to end our functions with `print`, how do we send our results to the screen?" The answer to that question is so straightforward that it's sometimes hard to see: We just make `m` be the last line of the function, like so:

```
(defun change-m (m)
    .
    .
    .
    m)
```

Remember that *any* AutoLISP object can be a form that is handed to `eval` for evaluation, including a symbol. When we give `eval` a symbol, `eval` returns the symbol's value. Therefore, if we simply make the last line of our `defun` be `m`, `eval` returns `m`'s value and it is printed *once* at the Command: prompt.

It might seem strange to see a lonely symbol as the last line of our `defun`, especially to those of us who have programmed in other languages. But it is entirely valid in AutoLISP and, in most cases, the preferred way of coding. In general, we write functions that *return* rather than *print* their result.

While we're on this subject, here's something else to watch out for. New users sometimes don't believe that we can make a symbol be the last form in the body of a `defun`; they feel better putting the symbol in parentheses, like so:

```
(defun change-m (m)
    .
    .
    .
    (m))
```

When `eval` sees `(m)`, it recognizes it as a list. It labels `m` an "operator" and issues a "bad function" error, because the value of `m` is not a function.

The first object past an open parenthesis in any code that is evaluated must, Must, *MUST* be a valid operator, that is, a symbol whose value is a function. Never use parentheses arbitrarily; they have a specific meaning in AutoLISP.

5.1.4.2 The 1> Prompt

Typically, we don't write our defuns at the Command: prompt, but nothing prevents us from doing so. In that case a function definition might appear as follows:

```
Command: (defun add-3 (n)
1> (+ n 3
2> )
1> )
ADD-3
```

Every time we hit <enter> in the middle of an expression, the reader tells us the number of unbalanced parentheses in that form. 1> tells us we have one open parenthesis that is missing its mate. 2> informs us that we have two unbalanced open parentheses, and so forth. This notation applies not only to defuns, but to any form we enter at the Command: prompt.

We usually avoid this prompt by entering the appropriate number of closing parentheses. Sometimes, however, we get this prompt because we entered a form incorrectly. Then we simply press <esc> to return to the Command: prompt. Here's another way we can get the 1> prompt:

```
Command: (setq s "Some string)
1>
```

In this case we forgot the ending double quote on the string. Because of this, AutoLISP assumes that the close parenthesis is part of the string. Even if we enter a right parenthesis thereafter, AutoLISP still assumes we're building a string:

```
Command: (setq s "Some string)
1> )
1>
```

One way to exit this condition is to enter a double quote to end the string and a right parenthesis to end the form:

```
Command: (setq s "Some string)
1> ")
"Some string)\n"
```

When we do this, we end up with garbage in the string. For this reason, when we forget a string's closing double quotes, it's easier to just enter <esc> and retype the expression. Note also that once we hit the <enter> key at the Command: prompt, we cannot go back to a previous line; if we make a mis-

take, we must retype the entire form. This is another important reason that we usually write `defuns` with an editor . . . it enables us to easily make corrections to our code.

5.2 Global and Local Variables

As we have seen, symbols are used extensively in AutoLISP to name objects that don't otherwise have names and as variables to hold temporary values. They are frequently categorized further as being global variables or local variables.

Global variables are symbols that are known to all functions in the AutoLISP world. Entering or exiting a function has no effect on global variables (although they do disappear when we close the drawing). When we type

```
Command: (setq x 3)
3
```

`x` becomes a global variable. In fact, unless we do something to specifically make a variable local, it is global by default. Until now, most of the variables we have seen are global variables.

Local variables are known only within the function in which they are defined. They're created when the function is entered and given values within that function. When the function is exited, the variables disappear.

We can create symbols just by typing them in. If we enter

```
Command: !xyz
nil
```

AutoLISP creates an unbound symbol named `xyz`. We don't do this very often because it isn't useful. We more commonly create a symbol via `setq` so we can give it a value:

```
Command: (setq xyz 6)
6
```

Whenever we create a symbol in this manner it is global.

At first we might think that global variables are better. If we can access a variable from *any* function that we write, our code becomes far more flexible. However, using globals has a couple of disadvantages:

- It is much easier to follow a program's flow if the variables are neatly defined in the functions where they are used. If we use globals in all of our functions, it becomes difficult to learn how certain variables obtain their values and how they are used.

  Suppose we alter the value of the global variable `y` within several functions. At some point `eval` is given the code `(/ x y)` and returns a "divide by zero" error. Somewhere in our code `y` was accidentally bound to 0. But where? Since `y` is a global variable, any function might have set its value. The extra flexibility that globals provide sometimes carries the added weight of more difficult debugging. On the other hand, globals do occasionally make debugging easier, as we'll see shortly.

- The other problem incurred by the use of global variables is even more egregious. Examine the following function:

```
(defun clearx ()
  (setq x 0)
)
```

  Suppose we give this function to someone else at work. Maybe it's part of a group of functions that perform a complicated task. What happens if a colleague is using the variable `x` to save the result of some calculation that she is performing? When our code changes the value of the global variable `x`, our friend's saved value is lost . . . and that colleague may no longer be our friend!

  The point is that we must be very careful in our use of global variables because they can have disasterous effects on someone else's code. Actually, we're more inclined to clobber our own code than someone else's, because programmers have a tendency to repeatedly use the same variable names. When we use globals in a function, it is an important courtesy to document that fact via comments before the code.

Many AutoLISP coders use global variables almost exclusively. Globals are easier to use because, being the default, they require no setup overhead. Paradoxically, local variables are almost always the proper tool.

5.2.1 Creating Local Variables

We can create local variables in a `defun` in two ways. The first is via parameters.

> A function's parameters are automatically local in that function.

Look again at the add-em function:

```
(defun add-em (n1 n2)
  (+ n1 n2))
```

The parameters n1 and n2 are local to the add-em function. They are given values when the function begins, and they go away when the function ends. To see how this works, let's enter the following code (remember, we can set the values of multiple variables in one setq):

```
Command: (setq n1 5 n2 6)
6

Command: (add-em 3 4)
7
```

Prior to running the function we set the values of global variables n1 and n2 to 5 and 6, respectively.

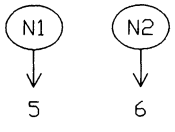


Figure 5-15

When we call the add-em function, one of defun's tasks is to save the parameters' global values, then rebind the parameters to the arguments.

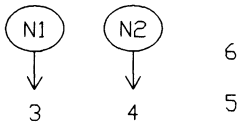


Figure 5-16

Within the above call to add-em, n1's value is 3. We can do anything we want to n1, including binding it to another object with setq:

```
(defun add-em (n1 n2)
  (setq n1 (+ n1 n2)))

Command: (add-em 3 4)
7
```

In this version of add-em, n1's value is changed, but n1 is still local to the function.

When the function completes its task, the last thing that `defun` does before returning control to us is rebind `n1` and `n2` to their original values:

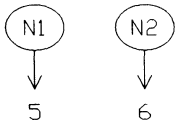


Figure 5-17

If we now examine the value of `n1`, we see that it is once again 5:

```
Command: !n1
5
```

This is an astonishing fact when we first encounter it. Please take the time to type in this code and verify for yourself that this is really how AutoLISP works.

Local variables are an important protection mechanism. Since `n1` and `n2` are local to the `add-em` function, we can give this function to anyone in our group and feel secure that even if they are already using variables with these names (local *or* global), our symbols won't overwrite the values of theirs. Furthermore, we can always name our points `p1` and `p2`, and know that `p1` in one function will not interfere with `p1` in another function. This is true even if both functions are in the same file or if one function calls the other.

The following modification to the `add-em` function shows another situation where we'd want to use local variables:

```
(defun add-em (n1 n2)
  (setq n3 (+ n1 n2))
    .
    .
    .
)
```

In this example `n3` is bound to the sum of `n1` plus `n2`, then used in the `defun` for various calculations (not shown).

We don't want `n3` to be global because we're not interested in its value once the function has completed. We'd like it to be local so that it goes away and doesn't collide with another `n3` that we might be using in our program. Suppose we make `n3` a parameter:

```
(defun add-em (n1 n2 n3)
```

Now, when we call `add-em`, we must supply a third argument:

```
Command: (add-em 3 4 ?)
```

But what value do we supply? Whatever `n3`'s value is at the start of the function is irrelevant, because the first thing we do in the function's body is to change its value. It's ridiculous to require the user to always supply a third argument that's never going to be used.

Ideally, we'd like `n3` to be local because we only want to use it inside the function. But we don't want `n3` to be a parameter because we don't want the user to supply a value for it; we'll assign it a value within the `defun`. This situation arises quite frequently and is handled by the following modification to `add-em`:

```
(defun add-em (n1 n2 / n3)
```

Subsequent to all of the parameters in the parameter list we place a forward slash followed by as many additional local variables as we need for use within the function. We must separate our symbols and the forward slash by one or more spaces because / is a valid character in a symbol name. If we enter

```
(defun add-em (n1 n2/n3)
```

`defun` assumes that our second parameter is a symbol named `n2/n3`.

Situations frequently arise where we want to use local variables in a function that does not have any parameters. Perhaps we want `pt1` and `pt2` to represent points that the user enters in response to our program's prompts. In this case, we can write the forward slash without a preceding space:

```
(defun test (/ pt1 pt2)
```

A space is still needed to separate the forward slash from the symbol `pt1`.

Keep in mind that all symbols in the parameter list of a `defun` are local to that function. The ones before the forward slash are parameters and must be supplied values when we call the function. Those that follow the forward slash are local, but they aren't parameters; they cannot be supplied values when we call the function. Their values are `nil` until we use `setq` bind them. The following example shows both ways of creating local variables:

```
(defun door (width height / perimeter area)
  (setq perimeter (+ (* width 2) (* height 2)))
  (setq area (* width height))
    .
    .
    .
)

Command: (door 3.0 6.6)
```

`width`, `height`, `perimeter`, and `area` are all local to the `door` function. `width` and `height` are parameters; their values are provided (by arguments) when the function is called. When we call the function as shown above, `width` is bound to 3.0 and `height` is bound to 6.6 because there is a one-to-one, positional matchup between arguments and parameters.

We want `perimeter` and `area` to be local variables as well because we are going to use them within the `door` function only; once the function has ended we want these variables to go away. On the other hand, we don't want them to be parameters. Why should we require the user to supply this information when we can calculate it ourselves? For these reasons we put `perimeter` and `area` in the parameter list following the forward slash.

Creating local variables in this manner is not required but strongly suggested. If we don't put `perimeter` and `area` in the parameter list, the `door` function will run correctly. But when it finishes, these variables and their values will still be bound in the AutoLISP world.

5.2.2 Using Global Variables

Although it doesn't happen very frequently, we occasionally want to use global variables. Most often it is when we need to remember some result even after a function exits.

Perhaps we've written a function that asks the user to enter the length of a wire and saves this value in the variable `$length$`:

```
Length of wire: 16.8
```

The next time he runs this function we'd like to prompt him with the previous length as a default:

```
Length of wire <16.8>:
```

If `$length$` is local, it goes away when the function ends and we can't use it next time in the prompt. By making `$length$` global, it retains its value even after the function ends. This is analagous to the way the TEXT command works in AutoCAD. If we change the text height and rotation angle, TEXT remembers and prompts us with the new values the next time it is run. Global variables enable us to utilize this technique in functions that we write.

When we set global variables, we must be careful that they don't overwrite other globals with the same name . . . this is the problem that local variables alleviate. We can take two measures to ensure that this doesn't happen:

1. Give global variables unusual names that someone else (or we) won't accidentally choose. In the last example we named our global variable `$length$` for this reason.

2. Prior to any function that uses global variables, list the names of those variables in a comment.

Earlier we pointed out that the use of globals can cause debugging problems. In some instances, globals can actually aid our debugging efforts.

Sometimes our functions don't run properly because a variable's value is incorrectly set. If the variable is local, we cannot learn its value at the Command: prompt, because that variable is destroyed when we exit the function.

```
Command: !perimeter
nil
```

A global variable, on the other hand, retains its value and can be examined following the function's conclusion.

When we need to learn the value of a local variable, we can make that variable temporarily global. The easiest way to do that is to comment out the portion of the parameter list where we designate it to be local, by adding a close parenthesis and a semicolon, as shown underlined:

```
(defun door (height width);  / perimeter area)
```

Now we can enter `!perimeter` or `!area` to learn their values. When the program is debugged we simply remove the `);` and the variables again become local.

Additional aspects of global variables are discussed in Section 13.1.1.

5.2.3 Advanced Topic: Scoping of Variables

In the last section we discussed the differences between global and local variables. Now we'll look at a few more aspects of how local variables behave. If you are new to AutoLISP, these concepts are not important just yet. Come back to this section once you feel comfortable writing AutoLISP routines.

Let's write a trivial function called `tubing` with parameters named `x` and `y`:

```
(defun tubing (x y)
   x)
```

This function merely returns the value of `x`; we don't have a use for `y` just yet. As usual, the parameters `x` and `y` are local to the function.

At the Command: prompt we'll create a global variable, also called `x`, and assign it a value of 5:

Figure 5-18

```
Command: (setq x 5)
5
```

Next, we'll call the `tubing` function:

```
Command: (tubing 22.4 13)
22.4
```

Figure 5-19

When the `tubing` function is entered the global value of `x`, 5, is saved on a stack. The argument, 22.4, is assigned to `x` *for the duration of the function.*

When the function completes, the local value of `x`, 22.4, is returned, and `x` is reassigned its former value, 5:

```
Command: !x
5
```

This much we have seen. Now let's alter the body of tubing and create a new function, elbow:

```
(defun tubing (x y)
  (elbow y))

(defun elbow (x)
  (print x))
```

tubing calls the elbow function, and the latter prints the value of its parameter, x. The result is returned to the tubing function, which returns that value as well. (See Section 5.1.4.1 to learn why the result is printed twice.)

```
Command: (tubing 22.4 13)
13 13
```

The question of interest is, why is 13 printed rather than 22.4? Inside tubing, x has the value 22.4, just as before:

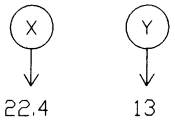


Figure 5-20

However, elbow is passed the value of y, which is 13. When elbow is entered, x's current value, 22.4, is saved away, and the new value, 13, is assigned to x.

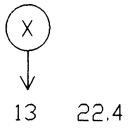


Figure 5-21

x's value, 13, is both printed by the elbow function and returned to the tubing function.

Don't be confused by our use of x for two different purposes in these functions. Remember that we can give our parameters any names we want; old values are saved away and later restored. x is local to each function and is assigned the value supplied by the caller.

A local variable named x that is defined in the calling function will not conflict with a local variable named x defined in the called function. However, the use of x to reference two different values in this manner is *NOT* recommended! As we can see, it makes code very difficult to read and understand. It was done here solely to illustrate AutoLISP's method of handling variables.

Let's alter our small example a little bit more:

```
(defun tubing (x y)
  (elbow))

(defun elbow ()
  (print x))
```

x is no longer a parameter in `elbow`, but the function still prints its value. Once again, we type `(tubing 22.4 13)` to run the function. What gets printed this time? Does `elbow` use the global value of x, 5, or does it inherit its value from `tubing`?

```
Command: (tubing 22.4 13)
22.4 22.4
```

At the time `tubing` calls `elbow`, the global value of x has been saved away:

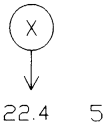


Figure 5-22

`elbow` inherits the local value of x from `tubing`; 22.4 is printed and returned.

In this example `elbow` makes a *free reference* to x, which means that the variable x is used in the `elbow` function but is not defined there. When a free reference is made to a variable, the most recent binding of that variable (i.e., its latest value) is used.

AutoLISP is a *dynamically scoped* language. This means that when `elbow` makes a free reference to x, the AutoLISP processor asks, "Who called `elbow`?" Since the calling function, `tubing`, defined x, its value for x is used. Were x not defined in `tubing`, the processor would have asked, "Who called `tubing`?" Since `tubing` is a top-level function, the global value of x (5 in our example) would then have been used.

Not all languages are dynamically scoped. Given the preceding example, some languages would simply use the global value of x when referenced from within elbow. Others just signal an error when a free reference is made.

Dynamic scoping lets us write more convenient and flexible code because we don't have to be totally strict about how we define our variables. However, when we nest functions many levels deep, it is good coding style to pass variables' values as parameters. Otherwise, bugs become much harder to track.

Example

Let's trace through an example that shows how data are passed to a function. The following function simply adds 2 to the values of a and b:

```
(defun add-2 (a)
  (setq b (+ b 2)
        a (+ a 2)))
```

We'll assign values of 10 and 20 to *global* variables a and b respectively, then call the add-2 function with the argument 5:

```
Command: (setq a 10 b 20)
20

Command: (add-2 5)
?
```

What does this function return (think about it, don't peek!)? What are the values of a and b at the conclusion of the function?

The symbol a is a parameter in the add-2 function, so it is local to that function. The symbol b, on the other hand, has not been defined locally, so it is a global variable. The function saves away the global value of a upon entry, and assigns a the value of 5; b retains its global value, 20.

2 is added to the value of b, yielding 22. 2 is also added to the value of a, making it 7. Since the latter is the last form evaluated in the body of the defun, a result of 7 is returned by the function:

```
Command: (add-2 5)
7
```

When the function is exited, a goes back to its initial value, 10. The global variable b was changed, however; its value is now 22:

```
Command: !a
10

Command: !b
22
```

5.2.4 AutoCAD System Global Variables

There are more than 100 AutoCAD system variables that we have read and write access to. Many of these, such as ANGBASE and OSMODE, are familiar to us; others are less well known. It's worthwhile scanning the table in Appendix A of the *AutoCAD Reference* to become aware of these variables; their settings can give us tremendous control over our drawing activities.

We can learn a system variable's value by using the `getvar` function.

`getvar` str	Takes the name of an AutoCAD system variable and returns its value.

For example, let's interrogate the LIMMAX variable to learn the upper right limits of our drawing:

```
Command: (getvar "limmax")
9.0 12.0
```

Note that the name of the variable *must* be specified as a string, but it doesn't matter whether the string is lower- or uppercase.

Unless a system variable is defined to be read-only, we can alter its value with the `setvar` function.

`setvar` str obj	Makes the object be the value of the AutoCAD system variable. It returns the object.

There are many useful variables whose values we can set. We can alter our drawing limits:

```
Command: (setvar "limmax" '(8 8))
8.0 8.0
```

We can change the fillet radius:

```
Command: (setvar "filletrad" 0.25)
0.25
```

We can adjust how often our drawing is saved:

```
Command: (setvar "savetime" 20)
20
```

Three switches that AutoLISP programmers frequently `setvar` are:

```
Command: (setvar "blipmode" 0)
0

Command: (setvar "highlight" 0)
0

Command: (setvar "cmdecho" 0)
0
```

These variables are initially set to 1; we set them to 0 to turn them off.

When we select an object on the screen, a little cross, called a *blip,* appears where we click. Eventually our screen gets filled with blips and we have to do a REDRAW. If our program (rather than the user) selects the points, there's really no reason to have the blips drawn on the screen. Thus, we set BLIPMODE to 0 to turn off this process.

Similarly, when our program chooses numerous entities, there's no gain to highlighting them; that just slows down the program. Therefore, we frequently turn off highlighting as well.

When we run an AutoCAD command such as LINE, it prompts us for the information that it needs. A typical LINE command might look like this:

```
Command: line
From point: 3,4
To point: 5,7.2
To point: 8.8,4.1
To point: c
```

If we supply points to the LINE command from within our function we have no need for the "To point" prompts. In fact, they'd just scroll up our screen's text area very quickly and surprise anyone using our function. We can disable

AutoCAD's prompts by setting CMDECHO to 0. This makes our AutoLISP functions run more cleanly.

If we want to turn on a feature that we have previously disabled, we can simply `setvar` any of these variables back to 1. Regardless, whenever we enter a drawing these variables are initialized to 1. Some programmers like these features to always be turned off and place the above `setvars` as a script in their `acad.lsp` files. When doing so, we typically set these inside of an `s::startup` function.

In Section 9.4 we see an elegant method of saving and restoring the values of system variables.

5.2.5 Review Topic: Using Symbols in AutoLISP

One of the hardest parts of learning AutoLISP is keeping straight the definitions of all of the terms that we use, especially as concerns symbols. First, we frequently use the words *symbols* and *variables* interchangeably. Then we talk about *global* variables and *local* variables. On top of that, symbols seem to have other names at different times: *operators, operands, arguments, parameters.* Let's review the differences among these.

Symbols

A *symbol* is an AutoLISP object type. We always draw symbols with circles around them. Except for strings, they are the only AutoLISP objects whose printed representations look like English-language words. But since strings *must* be contained within double quotes (e.g., `"Enter a width"`), they are easily differentiated from symbols. Symbols are *always* printed by AutoLISP in uppercase. Therefore, the following rule applies:

> When AutoLISP prints an uppercase word that is not within double quotes, that word must be a symbol.

Symbols can have special characters in their names. For example, + is a symbol. `$%##` can be too. But these are exceptions rather than the rule.

Symbols are used to *name* things. Many AutoLISP objects, such as lists and functions, don't have names. Therefore, we make these objects be the values of symbols so we can access them. When we supply a symbol to `eval`, it returns

the symbol's value, giving us the object that we want. Typically, we're far more interested in the object that the symbol is bound to than the symbol itself.

Variables

We call a symbol a *variable* when it is used in an algebraic manner to represent an unknown quantity. In algebra (or FORTRAN or BASIC for that matter); we might say

```
x = 3
```

We accomplish the same thing in AutoLISP by entering

```
(setq x 3)
```

In either case, x is a variable that has been assigned a value of 3.

We frequently use the term *variable* to indicate a symbol that references some data. An example of a symbol that would *not* be considered a variable is a function name. Consider the add-em function:

```
(defun add-em (n1 n2)
  (+ n1 n2))
```

When loaded into the AutoLISP world, this function becomes the value of the symbol add-em. add-em is considered to be a function name, not a variable. n1 and n2, on the other hand, are variables.

Operators and Operands

Once we load the add-em function we can invoke it in the following manner:

```
Command: (setq x 3 y 4)
4

Command: (add-em x y)
7
```

First we set the values of the variables x and y to 3 and 4, respectively. Then we pass these values to add-em, which adds them together and returns their sum, 7.

In the form (add-em x y), add-em is the *operator* and x and y are *operands*. Remember from our discussion of prefix notation that the first symbol after the open parenthesis is called the operator and all the other elements of the list are operands.

The operator is *always* a symbol whose value is a function. Putting it differently, a function name *must* be the first element of any list that is evaluated. If it's not, we get a "bad function" or "null function" error.

Operands can be any AutoLISP object, but we must make sure that they evaluate to datatypes that are acceptable to the function.

The operator and operands are what we see in a printed representation; arguments usually don't appear there.

Arguments

We call the result of the evaluation of an operand an *argument.* When we hand the form `(add-em x y)` to `eval`, one of `eval`'s tasks is to evaluate each of the operands. It evaluates the operands `x` and `y` to get the arguments, 3 and 4. Note that in the preceding example the arguments aren't symbols; like operands, arguments can be any AutoLISP object.

Functions only use arguments; each operand must be evaluated. Some AutoLISP objects, such as numbers and strings, are *self-evaluating:* when they are given to `eval`, `eval` simply returns them. When this happens we *do* see the arguments in the printed representation of the form.

Parameters

Parameters are symbols that exist only inside of `defuns`. The first thing in a `defun` following the name of the function is a list of the parameters that the function uses. A parameter is a placeholder for data that are unknown at the time we write a function but will be supplied when the function is called.

There is a one-to-one matchup between parameters and arguments. When a function is entered, AutoLISP binds each parameter to its corresponding argument. Each parameter maintains its value while the function executes (unless the function `setq`s it to something else), then goes away when the function finishes.

Global Variables

Global variables (or symbols) are those that are known to every function in our program. Whenever we `setq` a variable at the Command: prompt, we set the value of a global variable.

```
Command: (setq x 3)
3
```

In this example, x is a global variable; any AutoLISP function can access it.

Within a function every symbol that we use is global unless we take special actions to make it local.

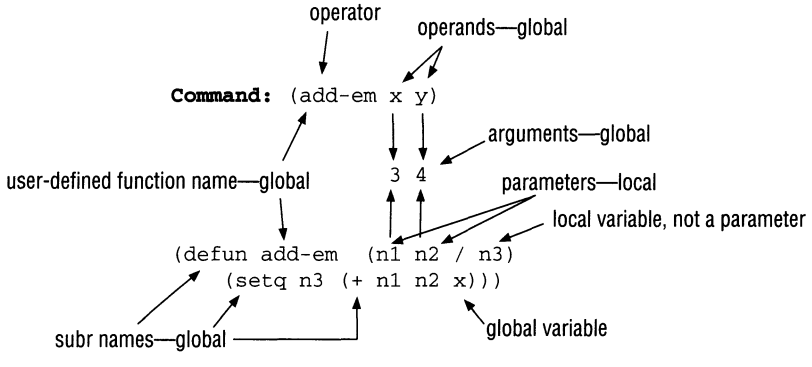
Local Variables

Local variables (or symbols) are known only within the function in which they are defined. They are created when the function is entered and destroyed when the function is exited. All of the variables in the parameter list of a `defun` are local to that function, but those following a forward slash aren't parameters.

Although local variables require extra work to set up, they are "cleaner" and safer than globals. Since they disappear when the function is exited, they enable two functions to use the same symbol names without the threat of having one variable accidentally changing the value of another.

Example

Suppose we've set the values of x and y to 3 and 4, respectively. The following picture shows the various uses of symbols:



5.3 Creating and Using AutoCAD Commands in AutoLISP

One of the main uses of AutoLISP is to automate tasks that we perform frequently in AutoCAD. As such, AutoLISP offers the following methods of interacting with AutoCAD:

- We can run most AutoCAD commands from within the AutoLISP functions that we write.

- We can create our own AutoCAD commands and even modify the existing ones.

5.3.1 Running AutoCAD Commands in AutoLISP

In addition to calling other functions within the body of a `defun`, we can also invoke most AutoCAD commands. Suppose we want to draw a line from 3,1 to 4,4. A problem arises if we enter the form

```
Command: (line '(3 1) '(4 4))
error: null function
```

`eval` returns a "null function" error because it assumes `line` is the name of an AutoLISP function we wrote rather than an AutoCAD command. We get an error when `eval` doesn't find a function named `line`. Furthermore, we can't simply put LINE inside of a `defun`, because AutoLISP will think it's a symbol and return its value (probably `nil`).

Instead, whenever we want to run an AutoCAD command in AutoLISP, we must run it inside of a subr named `command`, like so:

```
Command: (command "line" '(3 1) '(4 4) "")
nil
```

`command` str obj . . . obj	Takes the name of an AutoCAD command and runs that command. Other arguments are the data that the command needs. It returns `nil`.

The preceding `command` function causes the LINE command to run exactly as though we had typed it at the Command: prompt. Remember that in AutoLISP we represent a 2D point as a list of two real numbers. If we supply integers, AutoCAD automatically converts them to reals.

We must specify the command name, `"line"`, as a string. The subsequent arguments are responses to questions that the LINE command asks:

```
From point: 3,1
To point: 4,4
To point:
```

The null string, `""`, is equivalent to typing an <enter> on the keyboard, effectively ending the LINE command. We supply the null string wherever we'd type a <space> or <enter> to an AutoCAD command's prompt.

Immediately following the end of the LINE command we can issue another AutoCAD command:

```
Command: (command "line" '(3 1) '(4 4) "" "pline" '(5 5) '(7 3) '(8 4) "c")
nil
```

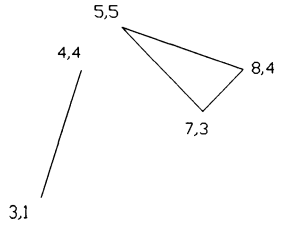


Figure 5-23

In this case we're following our line with a polyline from 5,5 to 7,3 to 8,4. `"c"` tells AutoCAD to close the polyline. There is no limit to the number of AutoCAD commands we can run within a given `command` function. However, it's usually clearer to issue a separate `command` function for each AutoCAD command we wish to run:

```
Command: (command "line" '(3 1) '(4 4) "")
nil

Command: (command "pline" '(5 5) '(7 3) '(8 4) "c")
nil
```

If you're typing this code as you read along, you're probably seeing a few `"To point:"` prompts. Recall that we can suppress these by entering

```
Command: (setvar "cmdecho" 0)
nil
```

It is occasionally inconvenient that the `command` function always returns `nil`, because we sometimes want to use the information that AutoCAD commands find. For example, the AREA command prints statistics on the screen, but we have no way of assigning those data to a symbol. For this command we can `getvar` the AREA and PERIMETER system variables.

We often have a choice concerning how we want to supply data to an AutoCAD command. For example, we can specify a point as either a quoted list: `'(3 1)`,

or as a string: "3,1". The latter is similar to the way in which we communicate with AutoCAD, and we're required to put it in double quotes (keyboard input to AutoCAD is generally specified as strings). Frequently we'll supply a point as the value of a symbol:

```
Command: (setq p1 '(3 1) p2 '(4 4))
(4 4)

Command: (command "line" p1 p2 "")
nil
```

Numeric values can be supplied as either numbers or as strings:

```
Command: (command "circle" '(5 6) 0.8 "circle" "3,4" ".8")
nil
```

This command function draws two circles. The first is centered at 5,6, the second is centered at 3,4, and they both have a radius of .8. Note that if we supply .8 as a number, the leading 0 is required. Otherwise we get an error:

```
Command: (command "circle" '(5 6) .8)
error: invalid dotted pair
```

Remember from our discussion of numbers that AutoLISP is not very robust in this respect. Without the leading 0 it thinks we're trying to enter a dotted pair (i.e., a cons) and doesn't recognize it as a number.

We can supply polar arguments to the command function:

```
Command: (command "line" '(1 1) '(6 2) "@2<90" "c")
nil
```

This draws lines from 1,1 to 6,2 to 6,4 (2 units 90 degrees from the X-axis), then closes them, like so:

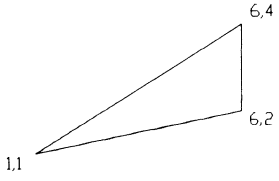


Figure 5-24

Some AutoCAD commands require input from the operator. We can indicate this within a command function by inserting the symbol pause. When the com-

mand processor encounters a pause, the command is suspended until the user inputs a value via the keyboard or a pointing device.

The following function draws a circle of radius 1.5 but prompts the user to supply the center point:

```
Command: (command "circle" pause 1.5)
nil
```

The CIRCLE command pauses until the user types in or clicks on a point, then proceeds to draw the circle.

Notes

- Backslash (\), the pause character in menus, works within the command function as well. However, Autodesk has reserved the right to change this in the future and warns that backslash should *not* be used within a command function. Incidentally, pause won't suspend menu input.

- When using pause, we typically set CMDECHO to 1 so that the operator will be prompted to enter the necessary data.

- We rarely issue the command function at the Command: prompt. Why bother? It's easier just to type the AutoCAD command. But command is an invaluable tool for use within functions we write.

- When the command function runs an AutoCAD command, we must supply data to that command exactly as if it were running at the Command: prompt. We're so familiar with these commands, however, that we often run them by rote and forget what information they actually request. Therefore, when adding a standard AutoCAD command to our functions, it's helpful to run that command once at the Command: prompt just to learn what questions it actually asks.

- Giving the command function no arguments is equivalent to pressing <esc> at the keyboard.

- Some AutoCAD commands cannot be used with the command function. These include DTEXT, PLOT, SKETCH, ZOOM DYNAMIC, and those with a DD prefix. command has several other minor idiosyncracies that are explained in the *AutoLISP Reference.*

Example

The following function is similar to AutoCAD's BOX command; it draws a box on the screen. The lines are numbered so that we may refer to them in our subsequent explanation of the code. The comments refer to the points supplied by the function call below.

```
1 (defun box (pt1 pt2)
2   (command "pline"                    ;Run the PLINE command
3     pt1                               ;From 2,3
4     (list (car pt1) (cadr pt2))       ;To 2,5
5     pt2                               ;To 4,5
6     (list (car pt2) (cadr pt1))       ;To 4,3
7     "c"))                             ;Close the polyline
```

Once written, we can call this function at the Command: prompt:

```
Command: (box '(2 3) '(4 5))
nil
```

When we call the box function, we *must* supply two arguments because the function has two parameters. The lists `(2 3)` and `(4 5)`, which represent the corner points of the box, become the values of the parameters, `pt1` and `pt2`. Thus, at the start of the box function, our data look like this:

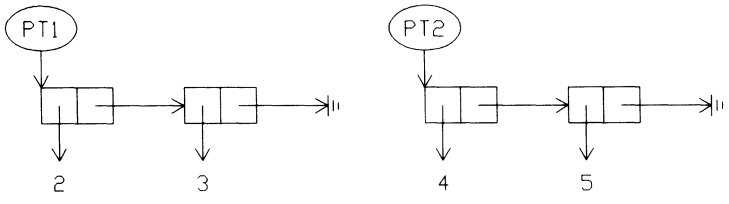


Figure 5-25

The entire box function contains only one form: a call to the AutoCAD POLYLINE command. As we know, we must supply PLINE points, and a 2D point is a list of two numbers. Here's what each line of code does:

1. The function's name is box; its parameters are `pt1` and `pt2`.

2. Call the AutoCAD PLINE command to draw a polyline connecting the following points.

3. The first point is the value of the variable `pt1`, which is the point 2,3.

4. The second point is a brand new list that we create with the `list` function. It is built from the car of `pt1`, which is 2, and the cadr of `pt2` (i.e., the Y co-ordinate), which is 5. The first polyline segment connects these two points:

Figure 5-26

5. The third point is the value of the variable `pt2`, which is the point 4,5:

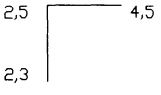


Figure 5-27

6. The fourth point is another new point comprising the car of `pt2`, 4, and the cadr of `pt1`, 3:

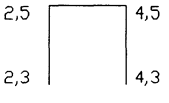


Figure 5-28

7. `"c"` tells AutoCAD to close the polyline, thereby finishing the box:

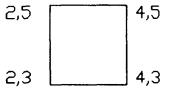


Figure 5-29

5.3.2 Creating AutoCAD Commands in AutoLISP

An important benefit of AutoLISP is that it enables us to create our own AutoCAD commands that are virtually indistinguishable from the built-in AutoCAD commands. We could say to someone in our group, "I've just received a command from the next AutoCAD release," and they'd have no way of telling that it's something that we wrote. We can place these commands in a menu or on our tablet, then use them exactly as we would any other AutoCAD command.

To create our own AutoCAD commands we write a `defun` in pretty much the same manner as we have seen thus far, but we tailor this `defun` to meet the criteria of an AutoCAD command:

- We must specify that what we're writing is an AutoCAD command rather than an AutoLISP function.
- The user must be able to run this command as he would a standard AutoCAD command. He won't place it in parentheses and he can't supply arguments.

  When we call, say, a LINE command, we don't supply the points at the time we invoke the command. Instead, we just enter LINE and the command prompts us for the data that it needs. Commands we write must work in the same manner.
- Our commands cannot have parameters. Since the user doesn't supply arguments when he calls the command, there is no need for parameters when we write it.

We can convert an AutoLISP function into an AutoCAD command by prefixing the name with `c:` For example, let's convert the `box` function we wrote in the previous section into an AutoCAD command, and name it `c:box2`:

```
1 (defun c:box2 (/ pt1 pt2)
2   (setq pt1 (getpoint "Enter corner point of box: "))
3   (setq pt2 (getcorner pt1 "\nEnter opposite corner: "))
4   (command "pline"
       pt1
       (list (car pt1) (cadr pt2))
       pt2
       (list (car pt2) (cadr pt1))
       "c"))
```

Once we have written and loaded the `c:box2` command, we can call it at the Command: prompt by entering

```
Command: box2
nil
```

Notice that, unlike a function, `c:box2` does not take arguments; in fact, we *cannot* provide data when we call this command. Furthermore, we don't put parentheses around the command name when we run it. Again, it must work *exactly* like a built-in AutoCAD command.

Although our AutoCAD commands cannot have parameters, the parameter list is required. We can leave it empty or we can specify as many local variables as

we need following a forward slash. Our `c:box2` command uses the local variables `pt1` and `pt2`, but they are not parameters.

Let's examine the `c:box2` command. The numbers below correspond to the numbers in front of the lines of code.

1. The `c:` specifies that this is an AutoCAD *command,* not an AutoLISP *function.* The `c:` has nothing whatsoever to do with the DOS c: disk drive specification. We must put `c:` in front of the name of every AutoCAD command that we write.

   The values of the local variables `pt1` and `pt2` are initially `nil`. They remain `nil` until we assign them values via `setq`. In this case we want to request points from the operator, just as AutoCAD does.

2. `getpoint` is a subr that halts the machine and requests that the user type in or pick a point. It waits until he does, then returns the selected point. `getpoint` is frequently used with `setq` because we want to save the point that was entered. In this example the selected point becomes the value of `pt1`.

3. `getcorner` is nearly identical to `getpoint`. It also waits for, then returns, a point selected by the user. In this case the point becomes the value of `pt2`. `\n` is used to force our text output onto a new line. In other words, it inserts a carriage return.

`get` functions are explained in Section 8.1.

   This is the first of several places in this book where we examine a program that contains subrs we've not yet seen. That's unavoidable if our examples are to be at all practical. When this situation arises, we'll explain enough about the subr for you to understand the example, then cover the subr in detail in its proper place in the book. Note that the subrs are all listed in the index.

4. Once we have received values from the user for `pt1` and `pt2`, the `c:box2` code runs the identical PLINE command that the `box` function used.

In Section 5.1.1 we saw how our functions can call other functions we have written. Our AutoCAD commands can call our own functions as well. `c:box2` is a good example of how we would do just that.

Notice that, starting with line 4, this command is identical to the body of the `box` function. Why bother retyping all of this code when we can just call the al-

ready-existing box function? In other words, we can rewrite c:box2 as follows:

```
(defun c:box2 (/ pt1 pt2)
  (setq pt1 (getpoint "Enter corner point of box: "))
  (setq pt2 (getcorner pt1 "\nEnter opposite corner: "))
  (box pt1 pt2))
```

When we run the c:box2 command, it prompts for the corner points of the box, then calls the box function to draw the box:

```
Command: box2
Enter corner point of box: 3,3
Enter opposite corner: 5,5        (A box is drawn on the screen)
nil
```

Note again that the command does not take parameters; instead, it prompts the user for the information that it needs. Thus it looks and runs exactly like a built-in AutoCAD command.

Here's an easy but powerful timesaver we can write in AutoLISP. Imagine that our boss has just given us a large drawing that needs massive updating ("Impossible," you say? Hah!). Perhaps the drawing has been scanned into AutoCAD and lines must be moved to match other lines. Thus, we're repeatedly moving a group of lines from the endpoint of one line to the endpoint of another. And to complicate matters further, we need to work with intersections too, and it's not convenient to set our running osnaps.

This isn't hard to do, but it requires an awful lot of cursor movements to the OSNAP menu. The project becomes a "snap" if we write a simple one-line AutoCAD command to run these steps for us:

```
(defun c:mov ()
  (command "move" "w" pause pause "" "end" pause "end" pause))
```

Whenever we type mov at the Command: prompt, this new AutoCAD command pauses for the user to enter the corner points of a window, then highlights the selected entities. It asks the user to select one of the lines and snaps to its endpoint. Finally, it prompts for the second line, then moves the selected entities to the endpoint of that line.

If we examine the work that we do daily, we'll discover certain AutoCAD tasks that we repeatedly perform. Now we know how to create our own AutoCAD commands to do most of that work for us.

Notes

- Because we're using this command exclusively at the Command: prompt, we probably want CMDECHO and HIGHLIGHT turned on.

- Most AutoCAD commands we write will request user input. We implement this with the `get` functions (such as `getpoint`), which are explained in Section 8.1. You might wish to read that section next so you can start writing useful AutoCAD commands right away.

- One question that arises is this: If commands do the same thing as functions but more nicely, why use functions at all? The answer is that commands and functions each have their niche, and we must know which is appropriate for a given situation.

  Commands are quite useful when we need to communicate directly with the user. They're easy to run because we don't need to put the command name in parentheses and because they prompt for the data that they need.

  On the other hand, prompts can interfere when we have no need to interface with the user. It's simpler from a programming standpoint to call a function and pass it data than it is to create the overhead of always asking the user for data. Furthermore, we often nest function calls several levels deep, passing down data that was calculated by the top-level function. In this situation we can't request the information from the user.

  One example of a nested function call is the `c:box2` command, which calls the `box` function. It would be inappropriate to make `box` a command because we want to pass data as arguments directly to the function. The distinction between the use of commands and functions will become more apparent as we gain familiarity with AutoLISP.

- We have seen that both our commands and our functions can call the AutoLISP functions we have written; `c:box2` calls `box`. It is also possible for our commands and functions to call other *commands* we have written. For example, we can call the `c:box2` command from within another command by entering `(c:box2)`, as follows:

```
(defun c: foo ()
    .
  (c:box2)
    .
    .)
```

5.4 Important Points to Remember

- We write functions to execute the same set of steps on different data.
- Give functions names that somehow reflect their purpose. Don't use the names of any of subrs or you'll redefine them. In particular, be careful not to reassign the `list` function.
- A *parameter* iş a symbol that is used as a placeholder for data that will be supplied when the function is called.
- Don't print a result on the last line of a function; have functions return results rather than printing them. Every function always returns a result back to whoever called it.
- Every time we correct a program using our editor, we must reload that program into memory. The editor puts a corrected version of our program on disk, but AutoLISP doesn't know about this version until we load it. We provide `load` with the name of the file, not the name of a function in the file. If the file contains only one function, however, it's usually best to give the file the same name as the function.
- Everything is assumed to be data until it is handed to `eval` for evaluation, whereupon it is assumed to be code.
- If we want an operand to be treated as code and evaluated, we cannot quote it. If we want an operand to be treated as data and not evaluated, then we must quote it. The decision is never arbitrary.
- Never quote either `defun`'s parameter list or a parameter itself.
- We create variables by typing them in. `setq` does not create variables, it just assigns them values. `setq` can bind both local and global variables.
- Use a global variable in a function to store data that will be needed after the function ends (perhaps it will be used the next time the function is called). Give the variable an unusual name so that it isn't accidentally overwritten, and put a comment describing it along with the function.
- All symbols in the parameter list of a `defun` are local to that function. The ones before the forward slash are parameters and must be supplied values when the function is called. Those that follow the forward slash are local

but they are not parameters; they cannot be supplied values when the function is called. Their values are `nil` until we use `setq` bind them. All other symbols we've seen so far are global.

- We must place one or more spaces between a symbol name and the forward slash in the parameter list. Otherwise the slash is considered to be part of the symbol's name.

- When we `defun` our own AutoCAD command we put `c:` in front of the command name. The command cannot have parameters but can have local variables following the forward slash. To invoke the command we type its name, without parentheses, at the Command: prompt.

- Figure 5-30 is the full `eval` flowchart, which includes the test for the special operator `quote`. `eval` actually has decision boxes that test for every special operator, because it must run special code to handle each of these. Get to know this flowchart really well; it's your key to understanding AutoLISP.

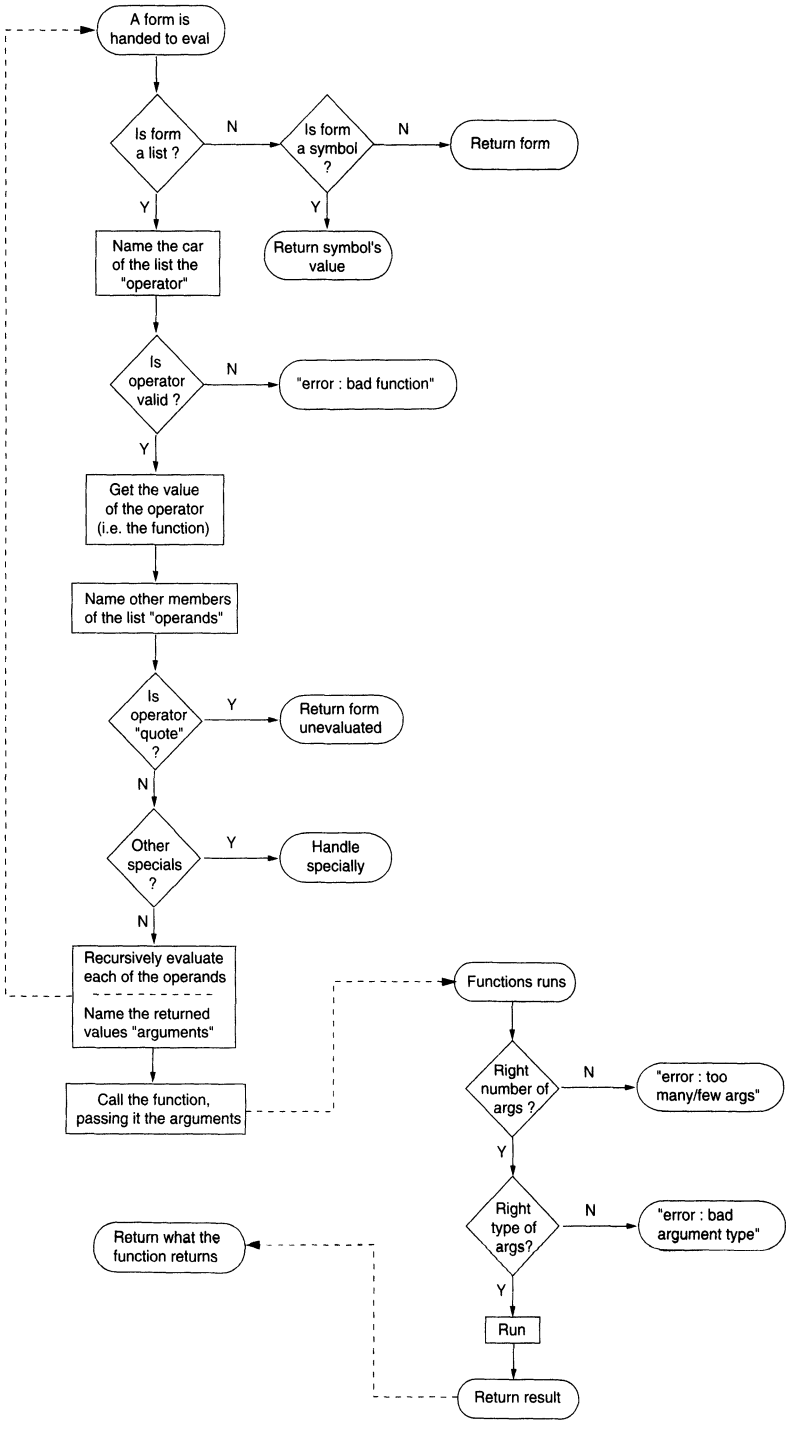

A form is handed to eval
Is form a list ?
N
Is form a symbol ?
N
Return form
Y
Y
Name the car of the list the "operator"
Return symbol's value
Is operator valid ?
N
"error : bad function"
Y
Get the value of the operator (i.e. the function)
Name other members of the list "operands"
Is operator "quote" ?
Y
Return form unevaluated
N
Other specials ?
Y
Handle specially
N
Recursively evaluate each of the operands
Name the returned values "arguments"
Call the function, passing it the arguments
Functions runs
Right number of args ?
N
"error : too many/few args"
Y
Right type of args?
N
"error : bad argument type"
Y
Run
Return result
Return what the function returns

Figure 5-30

5.5 Labs

The first exercise should be done at the Command: prompt and checked with the EFC and/or DRAW programs. Subsequent exercises that require you to create `defuns` should be written using an editor.

Note: For most exercises that require us to write a function, we will see one or two examples of how we would *call* the function once it has been written. These examples are not part of the function definitions, and the data they receive do *not* belong in the `defun`.

1. Set the values of `x` and `y` to 4 and 5, respectively. Then, for each of the following identify the operator, operands, arguments, and what is returned when we run the function. For an example, see Chapter 4, Lab 2.

   a) `(quote x)`
   operator:
   operand:
   result:

   b) `(list x y)`
   operator:
   operands:
   arguments:
   result:

   c) `(list x (quote y))`
   operator:
   operands:
   2nd level operator:
   2nd level operand:
   2nd level result:
   arguments:
   result:

   d) `(list x 'y)`
   operator:
   operands:
   2nd level operator:
   2nd level operand:
   2nd level result:
   arguments:
   result:

e) `(+ x 'y)`
operator:
operands:
2nd level operator:
2nd level operand:
2nd level result:
arguments:
result:

f) `(quote (x y))`
operator:
operands:
result:

g) `(quote x y)`
operator:
operands:
result:

h) `(list '(x y))`
operator:
operands:
2nd level operator:
2nd level operand:
2nd level result:
argument:
result:

i) `(list (x y))`
operator:
operand:
2nd level operator:
2nd level operand:
2nd level result:
arguments:
result:

j) `(list x '(y))`
operator:
operands:
2nd level operator:
2nd level operand:
2nd level result:
arguments:
result:

k) `(setq x 'y)`
operator:
operands:
2nd level operator:
2nd level operand:
2nd level result:
argument:
result:

l) `(setq a x b y)`
operator:
operands:
arguments:
result:

m) `(setq c (x + y))`
operator:
operands:
2nd level operator:
2nd level operand:
2nd level result:
arguments:
result:

2. As we saw in Chapter 3, if we `fix` a real number the fractional part is truncated, not rounded. If we want to round a positive number, we can add 0.5 to the number before fixing it. Write a function called `roundit` to accomplish this (ignore negative numbers). We'd run it as follows:

```
Command: (roundit 3.49)
3

Command: (roundit 3.5)
4
```

3. Write a function called `rtd` that converts radians to degrees. Divide the angle by `pi`, then multiply the result by 180.0. Model this function on `dtr`, which we wrote in Section 5.1.1.

```
Command: (rtd 3.141593)
180.0
```

4. Write a function that accepts a Fahrenheit temperature and returns its Celsius equivalent. The formula is `c = 5/9 * (f - 32)`.

```
Command: (ftc 212)
100.0
```

5. Write a function that takes a real number representing dollars, rounds it down to the nearest dollar (using the `fix` function), and calculates a 15 percent tip on this amount.

```
Command: (tip 10)
1.5

Command: (tip 10.75)
1.5
```

6. Write a function that takes a real number representing dollars and returns this amount with a tip added on. Have it call the `tip` function.

   This is a nice exercise because it requires us to write a function that calls another function we have written. Don't repeat the `tip` code in `add-tip`; have it actually call the `tip` function.

```
Command: (add-tip 10)
11.5

Command: (add-tip 10.75)
12.25
```

7. Write a function called `ins-str` that takes a string and a point as arguments and inserts the string into the drawing at the specified point. Hint: Run the TEXT command.

```
Command: (ins-str "A new string" '(4 6))     ;"A new string" is
nil                                          ;inserted at 4,6
```

8. Write an AutoCAD command called `c:ze` that performs a ZOOM EXTENTS.

```
Command: ze                  ;The drawing zooms to its extents
nil
```

   When this command works, write commands named `c:zp` and `c:zw` to perform ZOOM PREVIOUS and WINDOW, respectively. The latter should pause for the user to input the window's corner points. These commands are seeds for more intricate labs presented in subsequent chapters.

9. Write a function that takes a list as its only argument and returns the third element of that list. This function should work with *any* list of at least three elements.

```
Command: (third '(a b c d e))
C

Command: (third (list 2 4 6 8 10 12 14))
6
```

If you have tried hard to solve this exercise but can't quite figure out the best way to get started, see the detailed explanation in the Answers appendix.

10. Write a function that takes a list as its only argument and returns a portion of that list minus the first three elements. Assume that the list has *at least* four elements.

```
Command: (kill-3 '(a b c d e))
(D E)
```

11. Write a function that takes two 2D points and returns their midpoint.

```
Command: (midpt '(2 3) '(4 6))
(3.0 4.5)
```

12. Write a command called `c:endit` to save the drawing and exit. It should use the AutoCAD WBLOCK command to save the file without such information as unreferenced layer names, blocks, or named views. You may need to look up the WBLOCK command and the DWGNAME system variable in the *AutoCAD Reference*.

13. This exercise explores how AutoLISP manages local variables. Initialize the variables `number` and `total` to 22 and 1, respectively, then type in the `test` function. Prior to actually running the `test` function as shown below, try to determine what it will print on the screen. Also, once the function has been run, what will the values of `total` and `number` be?

```
(defun test (number)
  (setq total (+ total number))
  (setq number (+ number total))
  (print total)
  (print number)
  (print count))

Command: (setq number 22 total 1)

Command: (test 5)

Command: !total

Command: !number
```

14. This exercise assumes that you have read the Advanced Topic section on scoping of variables.

    First type in the following functions (it's probably best to use an editor, but these could be entered at the Command: prompt):

```
(defun p ()
  (print m)
  (print n)
  (q 9))

(defun q (m / n)
  (print m)
  (print n)
  (setq m 22)
  (r 4))

(defun r (n)
  (print m)
  (print n)
  (s))

(defun s ()
  (print m)
  (print n))

(defun v ()
  (print m)
  (print n))
```

Next, initialize m to 5 and n to 6:

```
Command: (setq m 5 n 6)
6
```

The defuns we typed are filled with print statements. List everything that will be printed by calls to p and v prior to running these functions.

```
Command: (p)

Command: (v)
```

III

Programming in AutoLISP

In Part III we examine the contracts of numerous AutoLISP subrs and learn how to put several functions together to build a program. We also study useful AutoLISP functions that are already written and write many of our own.

In Chapter 6 we learn AutoLISP's control structure. We see how to write conditional expressions that execute different pieces of code depending on whether a test succeeds or fails. This makes our programs flexible and useful in different circumstances. We also learn looping and recursion, which enable a function to operate on many entities in our drawing rather than only one, thereby greatly increasing the function's power and our productivity.

In AutoLISP, data are often stored in lists for ease of access and manipulation. In Chapter 7 we learn list-handling subrs by way of a fun quiz. You'll be surprised to see how much you can figure out without knowing the functions ahead of time. We also explore a couple of the trickier AutoLISP functions, `mapcar` and `lambda`, in an Advanced Topic.

One of the big benefits of AutoLISP is the ability to write AutoCAD commands that are indistinguishable from the built-in ones. In Chapter 8 we learn how to make our own AutoCAD commands interface with the user in a polite and elegant manner. We also see how to read from and write to files so we can save data that we create in AutoLISP. A number of advanced I/O functions and procedures are covered as well.

By Chapter 9 we'll have a good understanding of how AutoLISP works and how to put functions together to build a program. In this chapter we learn the contracts of numerous functions in order to expand our expertise in the language and further increase our productivity.

6

Control Structure: Producing Powerful Programs

Topics Covered in This Chapter

- Using predicate functions to run tests
- Conditional execution of code: `if`, `progn`, `and`, `or`, `not`, `cond`
- Writing loops: `repeat`, `while`, `foreach`
- Advanced Topic: Writing recursive functions

Goals for This Chapter

- Learn the special operators that provide control structure for our programs. Understand how they affect program flow, thereby unlocking AutoLISP's programming power.
- Learn how predicates are used as tests in conditional and iteration functions.
- Conditionals cause different blocks of code to be run, depending on the results of tests. Understand when to use each operator, its syntax, and how to set up the test.
- Iteration operators allow blocks of code to be executed a specific number of times or until some end condition is met. Learn which is the correct operator to use in a given situation and how to write a loop.
- Learn how to apply recursive solutions to a particular class of problems.

Introduction

This chapter provides a transition from the language fundamentals we have seen so far to the more practical aspects of AutoLISP. We begin by learning the contracts of many subrs. These subrs are integrally tied to the workings of AutoLISP because they enhance our control over the functions that we write.

The conditional and iteration sections introduce most of the remaining special operators in the language. The majority of them are special because they have one or more operands that may not be evaluated. There is special code in `eval` to handle each of these subrs.

The constructs shown in this chapter are common to most computer languages. If you've programmed before, these concepts will be familiar to you; you need only learn the syntax (i.e., the way in which it is written) of the individual subrs. If you're new to programming, it is essential that you gain an understanding of IF-THEN-ELSE expressions and how to write loops; almost every function we write will contain at least one of these facilities.

6.1 Predicate Functions: Asking Yes/No Questions

Predicates are functions that are used to ask questions of the AutoLISP world, usually of the yes/no or true/false variety. Most take one argument and run a test on that argument. They return `t` if the test succeeds and `nil` if it fails. Predicates are information-gathering functions only; they never change the AutoLISP world. We can write predicates that ask such questions as:

- Is the value of `x` greater than 0?
- Is this wire longer than the wire connecting those two terminals?
- Does this door fit in the door frame?

Predicates are used most often in conjunction with `if` and `while`, which are explained later in this chapter. We'll see numerous examples of predicates throughout the book and write our own as well.

`numberp` obj	Returns `t` if the argument is a number.

```
Command: (numberp 3)                    ;Integers are numbers
T
```

```
Command: (numberp 4.4)                ;Reals are numbers
T

Command: (setq n 6)
6

Command: (numberp n)                  ;The value of n is a number
T

Command: (numberp "33")               ;"33" is a string
nil
```

`listp` obj	Returns `t` if the argument is a list or a cons.

From its name we get the impression that `listp` checks for lists, but it's not quite that precise. It returns `t` if its argument is a *cons,* even if that cons is not a valid list.

`listp` also returns `t` if its argument is the symbol `nil`. Remember, `nil` is not only a symbol, it is also defined to be a list containing no elements.

```
Command: (listp '(a b c))             ;(a b c) is a list
T

Command: (listp '(a . b))             ;Dotted pairs are treated as lists
T

Command: (listp nil)                  ;nil is considered to be a list
T

Command: (listp "a b c")              ;"a b c" is a string
nil
```

`atom` obj	Returns `t` if the argument is an atom.

atom is a term used to refer to an AutoLISP object that is *not* a list or a cons. Lists and conses are *non-atomic* because they comprise other objects such as numbers. Object types such as symbols, strings, and numbers are *atomic* because they are not composed of other objects.

`atom` is the opposite predicate of `listp`. If an atom is supplied as an argument, `listp` returns `nil`. The one exception to this is `nil`, which is both an atom and a list.

```
Command: (atom 3)                    ;Numbers are atoms
T

Command: (atom 'x)                   ;So are symbols
T

Command: (atom "zzz")                ;Strings are atoms too
T

Command: (atom nil)                  ;Even nil is an atom
T

Command: (atom '(a b c))             ;Lists are not atoms
nil

Command: (atom '(a . b))             ;Nor are dotted pairs
nil
```

`null` obj	Returns `t` if the argument is the symbol `nil`. Otherwise it returns `nil`.

The `null` predicate is used specifically to test for the symbol `nil`. This can actually be a very helpful function because `nil` is used for so many different purposes, such as a placeholder in lists.

Suppose we have lists of data representing information about rooms in a house we're designing. One element of a list might be information regarding the windows in a room. If that room is an interior kitchen, we might use `nil` to indicate that there is no window information for that room.

Here's another example: If we're writing a loop that manipulates elements in a list, how do we know when we get to the end of the list? Well, the last cons has a cdr of `nil`. We can test for it with the `null` predicate.

```
Command: (null nil)                  ;Succeeds for nil
T

Command: (null t)                    ;Fails for t
nil

Command: (null 'jkl)                 ;Or anything other than nil
nil

Command: (null jkl)                  ;jkl is unbound
T
```

A practical example using `null` is shown with recursion in Section 6.4.

boundp obj	Returns `t` if the argument is a symbol with a non-`nil` value. It returns `nil` if the argument is either a symbol whose value is `nil` or an object other than a symbol.

When we create a symbol in AutoLISP its value is `nil` until we assign it a value with `setq`, whereupon it is said to be *bound* to that value. We can use the `boundp` predicate to determine whether a symbol is indeed bound:

```
Command: (setq x 3 y nil)
nil

Command: (boundp 'x)            ;x is bound; its value is non-nil
T

Command: (boundp 'y)            ;y is not bound; its value is nil
nil

Command: (boundp 'z)            ;z is unbound until we assign it a value
nil

Command: (boundp x)             ;The value of x is 3, not a symbol
nil
```

When testing whether a symbol is bound, remember to quote the symbol!

In practice, we more frequently test whether a symbol's value has *become* `nil` than test whether it was bound in the first place. In Section 13.1.3 we will see a more efficient way of testing whether a symbol is bound, thereby eliminating the need for `boundp`.

zerop num	Returns `t` if the argument is 0.

```
Command: (zerop (* 5 0))        ;Result is 0
T

Command: (zerop 7.4)            ;Not 0
nil
```

`zerop` differs from many of the other predicates in that its argument *must* be a number:

```
Command: (zerop 'x)             ;Not a number
error: bad argument type
```

minusp num	Returns t if the argument is less than 0.

```
Command: (minusp (- 3 8))              ;(- 3 8) is negative
T

Command: (minusp 6)                    ;6 is not negative
nil
```

minusp is like zerop in that its argument *must* be a number:

```
Command: (minusp 'x)                   ;x is not a number
error: bad argument type
```

minusp can be used in conjunction with the not function to test for positive numbers:

```
Command: (not (minusp 6))              ;6 is positive
T

Command: (not (minusp (- 3 8)))        ;(- 3 8) is negative
nil
```

type obj	Returns a symbol that indicates the datatype of the argument.

The type function is not strictly a predicate because it doesn't return t or nil. However, it tests its argument in a manner comparable to the predicate functions and is used in similar applications.

type is one of the few functions that returns a symbol as its result. Following is a list of symbols returned by type, along with a description of each symbol:

REAL	Floating point number
INT	Integer
SYM	Symbol
STR	String
LIST	List or user-written function
SUBR	Built-in function
EXSUBR	External function (ADS)
FILE	File descriptor
ENAME	Entity name
PICKSET	Selection set
PAGETB	Function paging table
nil	The symbol nil

The following examples show how `type` tests for the datatypes that we have thus far encountered:

```
Command: (setq a 5)
5

Command: (type 'a)                    ;a is a symbol
SYM

Command: (type a)                     ;The value of a is an integer
INT

Command: (type (float a))             ;If we float a we get a real
REAL

Command: (type '+)                    ;+ is a symbol
SYM

Command: (type +)                     ;The value of + is a subr
SUBR

Command: (type "abc")                 ;"abc" is a string
STR

Command: (type '(a b c))              ;(a b c) is a list
LIST

Command: (type '(a . b))              ;(a . b) is also considered a list
LIST                                  ;  even though it's a cons

Command: (type nil)                   ;nil is both a list & a symbol
nil                                   ;  so type just returns nil
```

6.1.1 Equality and Comparison Predicates

Suppose we create the following cons:

```
Command: (setq x '(a . b))
(A . B)
```

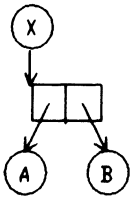


Figure 6-1

Next we enter the following code:

```
Command: (setq y '(a . b))
(A . B)
```

Will this create a brand new cons or will it just reuse the cons we previously created?

The answer is that a new cons is constructed each time we type one in. We could have created either cons with the `cons` function:

```
Command: (setq y (cons 'a 'b))
(A . B)
```

But no matter how we specify it, the `cons` function builds a new cons. On the other hand, it reuses the symbols `a` and `b`:

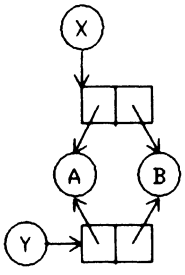


Figure 6-2

The next question is, does the following code create a new cons?

```
Command: (setq z y)
(A . B)
```

No, it does not. In this case we are not building a new cons, we're merely saying, "Make the value of `z` be the same as the value of `y`, whatever that might be." Thus, the following relationship is created:

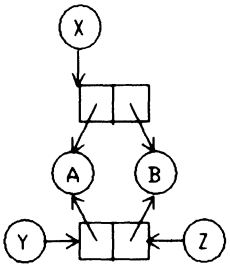


Figure 6-3

We have now bound three symbols to two different conses, but the values of each of these symbols have an identical printed representation:

```
Command: !x
(A . B)

Command: !y
(A . B)

Command: !z
(A . B)
```

This brings us to an important limitation of printed representation:

> Two totally different conses or lists may have the identical printed representation.

There is no way of telling from their printed representations whether two conses or lists are identical, or whether they just print the same. This is another limitation of printed representation that our picture notation clarifies. To distinguish among these objects, our code can use a predicate named eq:

eq obj obj	Returns t if the two arguments are *exact same* AutoLISP object.

```
Command: (eq x y)          ;x and y look the same
nil                        ; but aren't the same object

Command: (eq z y)          ;z and y are same object
T
```

In our example, the value of x and the value of y are not identical AutoLISP objects, even though their printed representations are the same. Therefore, they are *not* eq.

The symbols y and z, on the other hand, have the exact same values (as Figure 6-3 clearly shows, their arrows point to the same cons). Thus, their values *are* eq.

Now consider the following examples:

```
Command: (eq 'a 'a)        ;There is only one symbol a
T
```

```
Command: (eq 3 3)                ;There is only one number 3
T

Command: (eq "abc" "abc")        ;There is only one string "abc"
T
```

Two atoms that appear the same are always eq.

There is another function named `equal` that works quite similarly to `eq`:

`equal` obj obj {num}	Returns `t` if the two arguments appear the same (i.e., have the same printed representation). The optional third argument specifies the amount by which the first two arguments can differ and still be considered equal.

Given the relationships shown in Figure 6-3, we can use `equal` to examine the values of the variables:

```
Command: (equal x y)             ;x and y print the same
T

Command: (equal y z)             ;So do y and z
T
```

In this example the value of `x` and the value of `y` are equal. They have the same printed representation even though they are not identical AutoLISP objects.

Since the values of `y` and `z` are eq, they perforce print exactly the same. Therefore, they are equal as well. In fact, any two objects that are eq must also be equal. `eq` is a more rigorous test.

Two atoms that appear the same are always both eq and equal. On the other hand, two conses or two lists that appear the same are equal, but they are not necessarily eq.

What this means is that for any AutoLISP object except conses and lists, `eq` and `equal` are completely interchangeable. In practice, it usually doesn't matter which function we use to test conses and lists either. However, it's occasionally necessary that we do choose the proper function, so it's important to understand the distinction between them.

equal has an optional third argument, called the *fuzz factor.* The fuzz factor tells AutoLISP the maximum amount that two numbers or two points can differ and still be considered equal. For example, if the value of m is 1.23 and the value of n is 1.25, then

```
Command: (equal m n)                    ;Not exactly equal
nil

Command: (equal m n .021)               ;Equal within .021
T
```

The use of fuzz is sometimes necessary because floating point arithmetic tends to be slightly inaccurate.

The fuzz factor also comes in handy when comparing points. As we know, the UNITS command allows us to limit the amount of digits that AutoCAD displays to the right of a decimal point. When we print a point, we often don't see all of the digits in each of its coordinates.

In comparing two seemingly identical points, equal returns nil if the points differ, even though they appear the same on the screen. We can use the fuzz factor to allow very close points to be considered equal. For example,

```
Command: (setq p1 '(3.0 4.0000001))
(3.0 4.0)

Command: (setq p2 '(3.0 4.0000002))     ;These appear equal
(3.0 4.0)

Command: (equal p1 p2)                  ;They're not
nil

Command: (equal p1 p2 0.001)            ;But they almost are!
T
```

equal compares the X, Y, and Z coordinates of each point in turn. It does not measure the total distance between the two points:

```
Command: (setq p1 '(3.0 4.0))
(3.0 4.0)

Command: (setq p2 '(3.7 4.8))           ;X coordinates differ by 0.7
(3.7 4.8)                               ;Y coordinates differ by 0.8

Command: (equal p1 p2 0.9)              ;Each is within 0.9
T
```

In this example, the Y coordinates of the two points are 0.8 apart. Since the difference is less than the fuzz factor, `equal` returns `t`. It doesn't matter that the distance between the two points is greater than 0.9.

Advanced Note

The fuzz factor has an inconsistency that you should know about if you plan to use this feature frequently.

The *AutoLISP Reference* states that the fuzz factor specifies "the maximum amount by which expr1 and expr2 can differ and still be considered equal." In practice, if the fuzz factor is *exactly equal* to the difference between the first two numbers, it sometimes returns `t` and other times returns `nil`.

The following examples demonstrate this behavior, but you might find that they return different results on your machine.

```
Command: (equal 4.0 4.3 0.3)          ;These are exactly 0.3 apart,
T                                     ; so t is returned

Command: (equal 4.0 4.2 0.2)          ;But this should also return t
nil
```

To be certain that the fuzz factor will work, we should make it a little larger than the expected difference:

```
Command: (equal 4.0 4.2 0.2001)       ;This ensures the correct result
T
```

The same inconsistency occurs when comparing points:

```
Command: (equal '(3.0 3.0) '(3.0 3.3) 0.3)    ;These Y coordinates are exactly .3 apart
T

Command: (equal '(1.0 1.0) '(1.0 1.3) 0.3)    ;But so are these
nil

Command: (equal '(1.0 1.0) '(1.0 1.3) 0.301)  ;This ensures the correct result
T
```

♦ ♦ ♦

Let's look at another aspect of `eq` and `equal`. What do you suppose the following functions will return?

```
Command: (eq 3 3.0)

Command: (equal 3 3.0)
```

Are 3 and 3.0 the exact same AutoLISP object? No. Do they have the same printed representation? Again, no. Yet, surprisingly, AutoLISP finds them to be both eq and equal. Except when the fuzz factor is involved, `eq` and `equal` are not the functions to use for comparing numbers and strings. For that task we use =.

= obj. . .obj	Returns `t` if all of the arguments are numerically equal.

The = (called "equal sign") function is intended for use with numbers and strings. We use it in the following manner:

```
Command: (setq m 3)
3

Command: (= 3 3.0 m)                ;All three are the same
T

Command: (= "a" "a")
T

Command: (= "a" "A")                ;Upper and lowercase strings
nil                                 ; are not =

Command: (= '(1 2) '(1 2))          ;Works like eq here
nil
```

= has an advantage over `eq` and `equal` in that it accepts as many arguments as we choose to supply (and it requires less typing!). For that reason many programmers use = most of the time, in lieu of `eq` or `equal`.

= works like `eq` when we give it lists, but that's not its intended purpose. = should be used with numbers and strings only; it's not really defined for other object types, and the results it returns aren't guaranteed. We use `eq` or `equal` when comparing any datatypes except numbers or strings.

= is from a family of comparison subrs. The following functions all take an unlimited number of arguments and return `t` if the comparison succeeds and `nil` otherwise.

/= obj. . .obj	"Not equal" returns t if any argument is not equal to the other arguments.

```
Command: (/= 3 3.1 (/ 9 3))              ;3.1 is not the same
T
```

< obj. . .obj	"Less than" returns t if its arguments are in strictly ascending order.

```
Command: (< 2 4 6 8)
T

Command: (< 2 3 3 4)                     ;3 is not less than 3
nil

Command: (< "A" "C")                     ;The ASCII value of "A"
T                                        ; is less than "C"
```

<= obj. . .obj	"Less than or equal" returns t if each succeeding argument is less than or equal to the previous argument.

```
Command: (<= 2 3 3 4)                    ;3 is less than or equal to 3
T

Command: (<= 2 3 2.9999 4)
nil
```

> obj. . .obj	"Greater than" returns t if its arguments are in strictly descending order.

```
Command: (> 93 19 8 3 -4)
T

Command: (> 18 6 6 7)                    ;6 is not greater than 6
nil
```

>= obj obj	"Greater than or equal" returns t if each succeeding argument is greater than or equal to the previous argument.

```
Command: (>= 8 6 6 5 4)                    ;6 is greater than or equal to 6
T
```

♦ ♦ ♦

Here's a very handy application of these comparison subrs: They can be used to test whether a number or string falls within a specific range:

```
Command: (setq num 56)
56

Command: (< 0 num 101)
T
```

The preceding function asks, "Is 0 less than the value of num and is the value of num less than 101?" In other words, we are testing whether the value of num falls in the range 1 to 100. Since the answer is yes, < returns t. Most computer languages require two steps to ascertain this information.

The next example tests whether the value of the variable char is in the range "a" to "z" inclusive, that is, whether it is a lowercase alphabetic character:

```
Command: (<= "a" char "z")
```

In the next section we use this code with the or function in a practical example.

Notes about Predicates

- Predicates often end in the letter *p:* numberp, listp, zerop. This indicates to someone reading our code that the function is a predicate. When we write our own predicates, it is good programming style to have their names end in the letter *p* as well.

- The and, or, and not subrs also return either t or nil but are not considered to be predicates; they are used to control the execution of our code. These special operators are discussed in the following section.

- Another predicate, wcmatch, works with strings and is defined in Section 9.1.1.

6.2 Conditional Execution of Code

Much of the power of any programming language lies in its ability to execute different code based on conditions that are present at the time a function is run. For example, we might test to see whether a door fits into a door frame. If it does, our program will place it there. If not, it will first size the door to fit the frame, and then place it there. Some doors will fit their frames and others won't. Our code has the capacity to perform an operation based on the condition of each individual door.

The main conditional function, which you'll recognize if you have programmed before, is `if`. It's conceptually identical to `if` functions in other languages, but we must learn its AutoLISP syntax. The other conditionals are either used with `if` or are variations of this function. They all conditionally execute code based on the results of one or more tests. The tests are frequently predicates but don't have to be.

Most of the conditionals are special operators because they have one or more operands that may not be evaluated. `eval` must determine that it has one of these special operators before endeavoring to evaluate the operands.

The iteration functions that we see in the next section are likewise special operators, because they too might have unevaluated operands. Excepting `lambda`, which is described in the next chapter, the conditional and iteration functions constitute the remaining special operators in the language.

`if` test form1 [form2]	If the test returns a non-`nil` result, form1 is evaluated. If the test returns `nil`, form2, if present, is evaluated. It returns whatever the last form evaluated returns.

The `if` special operator allows us to choose which of two pieces of code to evaluate based on the result of a test. This corresponds to the IF-THEN-ELSE structures present in most computer languages. The general format of `if` is as follows:

```
(if test
    then-form
    else-form)
```

`if` takes three operands:

- The first operand is a test. It's always evaluated.
- The second operand is evaluated (and third skipped) if the test returns non-`nil`. This is called the THEN path.
- The third operand is evaluated (and second skipped) if the test returns `nil`. This is called the ELSE path.

Notes

- The second operand is evaluated if the test returns anything except `nil`. This might be `t`, but it could just as easily be 5 or anything else. Only a test result of `nil` will allow the third argument to be evaluated. Because functions usually test for `nil` versus non-`nil`, `nil` is an extremely important symbol; `t` is less frequently used.
- Either the second or the third operand is evaluated, never both. For this reason `if` is a special operator.
- The third operand is optional. If it's omitted and the test returns `nil`, the `if` exits immediately and returns `nil`.

Here's an example using `if`:

```
(if (= total 100)
    (setq done-flag t)
    (setq total (+ total 1)))
```

This `if` tests whether the value of the symbol `total` is 100. If it is, the symbol `done-flag` is bound to `t`, then the `if` exits and returns `t`. If it's not, then `total` is incremented by one and its new value is returned as the result.

The test to `if` can be arbitrarily elaborate and nested as many levels as needed. For example, the following test examines the datatype of some object. If it's a string, then it's placed in the drawing at the point 3,3. The null string `("")` is equivalent to pressing `<enter>` in response to questions asked by AutoCAD.

```
(if (equal (type object) 'str)
    (command "text" '(3 3) "" "" object))
```

Notice that this is an IF-THEN; it has no ELSE path. If the test fails (i.e., returns `nil`), the `if` simply exits and returns `nil`.

`if` statements can be nested inside of other `if` statements. In other words, the THEN or ELSE path of an `if` can be yet another `if`. However, when that situation arises, we frequently use a `cond` function (described shortly) instead.

Example

Chapter 5, Exercise 2 had us write a function called `roundit` that takes a positive numeric argument and rounds that number to the nearest integer. If you haven't yet done so, you might wish to write that lab before reading this example.

A limitation of `roundit` is that it only works for positive numbers; we'd like for it to handle negatives as well. Here's how the function should work:

```
Command: (round -4.49999)
-4

Command: (round -4.5)
-5

Command: (round 4.5)
5
```

4.5 rounds up to 5, but –4.5 rounds down to –5. IF the number is positive THEN we round up, ELSE we round down. This function calls for the use of `if`.

As with `roundit`, `round` takes a number as its only argument:

```
(defun round (n)
```

We start by testing whether this argument is positive or negative. The predicate that enables us to do this is `minusp`:

```
(defun round (n)
  (if (minusp n)
```

If the argument is negative, then we must subtract .5 and fix the number. If the argument is positive, then we add .5 and fix it, like we did in the `roundit` function:

```
(defun round (n)
  (if (minusp n)
      (fix (- n 0.5))
      (fix (+ n 0.5))))
```

♦ ♦ ♦

AutoLISP's version of `if` has an important restriction. Based on the result of its test, `if` evaluates either the second or the third form in its body. In other words, `if` evaluates only *one form* following the test. Frequently we need to perform several tasks on either the THEN or ELSE path. As it is constructed, `if` is unable to evaluate more than one form in either case.

When we need to evaluate more than one form on either the THEN or the ELSE path of the `if`, we use another subr, `progn`.

`progn` body	Evaluates its body forms in order. Returns whatever the last form evaluated returns.

`progn` enables an `if` to evaluate more than one form on either its THEN or its ELSE path. It takes an arbitrary number of forms and evaluates them in order, returning whatever the last form in its body returns. `progn` is like BEGIN-END or DO-END in other languages. Its syntax is as follows:

```
(progn form
       .
       .
       .
       form)
```

The following `if` needs to perform two tasks if the test succeeds. Therefore, a `progn` is required:

```
(if (> counter some-limit)
    (progn (error-routine)
           (setq counter 0))
    (setq counter (1+ counter)))
```

If the value of the variable `counter` is greater than the value of the variable `some-limit`, the `if` takes the THEN path and evaluates the `progn`. The `progn` "tricks" the `if` into thinking it is performing only one task, that is, the `progn`. But within the `progn` we can specify as many forms as we wish.

In this case we want two tasks performed: first, an `error-routine` function (not shown here) is called. Then the value of `counter` is reset to 0.

Other languages would allow us to use parentheses to group the two tasks together. In AutoLISP, as we have seen, parentheses have a special meaning; their use for grouping purposes usually leads to an error.

If the test fails (i.e., the value of `counter` is not greater than the value of `some-limit`), the `if` takes the ELSE path and simply increments the value of `counter` by one. This path does not require a `progn` because only one form is being evaluated.

Aside: Notice the indentations that we use in this example. When `if`'s three operands are aligned in one column, we can easily see what the operands are. If we type all our code beginning in column 1, we can't visually identify the operands to any of the functions:

```
(if (> counter some-limit)
(progn (error-routine)
(setq counter 0))
(setq counter (1+ counter)))
```

The convention for indentation is to vertically align forms that are at the same level of parenthesis nesting. In our example, the functions `>`, `progn`, and the second `setq` are all in the second level of parenthesis nesting. Thus, they are indented from the `if` at the top level, and they align with each other.

Throughout this book we adhere to a rigid convention for parenthesis nesting. We encourage you to do the same to make your code more readable and maintainable. Some text editors will perform the proper indentations when you press the <enter> key. The AHED editor supplied on the floppy doesn't quite do this, but at least it heads us in the right direction.

and form . . . form	Evaluates its forms in order. If any form returns `nil`, and immediately exits and returns `nil`. If all forms return non-`nil`, and returns `t`.

The `and` special operator performs multiple tests or tasks as long as each one succeeds. Its syntax is as follows:

```
(and form
      .
      .
      .
     form)
```

`and` takes an unlimited number of operands and evaluates them in order. It is normally used as the test for `if`:

```
(if (and (> x 10)
         (zerop y))
    .
    .
    .)
```

If the value of x is not greater than 10, then the test returns nil and the and exits immediately, without evaluating any other forms in its body. If the value of x is greater than 10, then the next test is evaluated.

The second test asks whether the value of y is 0. If it isn't, the test returns nil and the and exits immediately, returning nil. Otherwise, both tests succeed and the and returns t. In order for and to return t, all forms in its body must return non-nil results. Because all of its body forms may not be evaluated, and is a special operator.

and can stand alone without the if:

```
(and (all-factors-checked-p)
     (no-errors-p)
     (princ "Testing complete, no errors."))
```

In this example, the first two forms are predicates that we've written beforehand. If either one returns nil, and exits immediately and returns nil. If and only if both functions return t, the third form gets evaluated, thereby printing the message indicating success.

A practical illustration of and without an if is shown in the example for while in Section 6.3. See Section 14.2.3 for an explanation of the AND concept.

or form . . . form	Evaluates its forms in order. If any form returns non-nil, or immediately exits and returns t. If all forms return nil, or returns nil.

The or special operator performs multiple tests or tasks until one succeeds. Its syntax is as follows:

```
(or form
    .
    .
    .
  form)
```

or takes as many operands as we choose to provide and evaluates them in order, as the following example shows. This example assumes that, prior to running the if, we bound the symbol char to a character entered by the user.

```
(if (or (<= "a" char "z")
        (<= "A" char "Z")
    (...)
    (prompt "Invalid input . . . try again"))
```

The if tests whether the value of the variable char is in the range "a" to "z" inclusive *or* "A" to "Z" inclusive. In other words, it tests whether the user entered a valid lower or uppercase alphabetic character.

If either test succeeds, if takes the THEN path and performs some useful work (not shown). However, if both tests fail, then the user did not enter a valid alphabetic character. In that case if takes the ELSE path and reprompts for input.

Because we use or, both conditions do not need to be satisfied (in fact, both *can't* be satisfied in this example) in order for the test to succeed.

As with and, or is typically used as the test to an if, but it can stand alone as its own function. Furthermore, we can nest ands inside of ors, and vice versa.

See Section 14.2.3 for an explanation of the Inclusive OR concept.

not obj	Takes any AutoLISP object as an argument and returns t if the argument is the symbol nil. Otherwise it returns nil.

The most common use of not is to reverse the test to an if or a while. In our discussion of if we saw an example in which we want to perform some action if the type of some object is a string:

```
(if (equal (type object) 'str)
    (command "text" '(3 3) "" "" object))
```

In this example, we evaluate the second argument if the test succeeds, but exit if the test returns nil. This is known as an IF-THEN expression.

Occasionally, we need to perform an action only if a test *fails*. This is called an IF-ELSE expression. Unfortunately, there is no mechanism in AutoLISP to support this concept. The way to implement an IF-ELSE is to use not to reverse the result of the test. if will take the THEN path if the test fails (i.e., returns nil) and exit if the test succeeds.

The following code tests whether the value of the variable num is equal to 0 prior to performing a division. We only want the division to be carried out if the test fails (i.e., the value of num is not equal to 0) so that we don't get a "divide by zero" error. not reverses the test result so that the function executes the division if the test fails and exits if it succeeds.

```
(if (not (zerop num))
    (/ total num))
```

not can also be used in conjunction with the minusp predicate. Suppose we want to ensure that the roundit function (Chapter 5, Lab Exercise 1) only rounds positive numbers. AutoLISP has no mechanism for testing for positives. Instead, we test for a negative number and reverse the test:

```
(defun roundit (n)
  (if (not (minusp n))
      (fix (+ n 0.5))))
```

Another place where not comes in handy is when we want to reverse the result of a test to the while function, thereby converting it into an until function.

♦ ♦ ♦

The final conditional operator that we'll study was actually the first one created in Lisp. It's name, cond, reflects the time when it was the only conditional in the language. In those early days clarity, consistency, and ease of use weren't really a consideration. Thus, although conceptually straightforward, cond has a very primitive and unwieldy syntax.

cond performs the same task as if. There is nothing that if can do that cond cannot; likewise cond can't do anything that if is unable to. Many new users find if to be a simpler and more intuitive function. However, instances frequently arise for which cond is actually more suitable than if. If you have programmed before, you've most likely encountered the following situation:

```
(if test
  then (...)
  else (if test
         then (...)
         else (if test
                then (...)
                else (...)
              )
       )
)
```

In this scenario we run a test. If the test succeeds, we execute a function and exit the `if`. Should the test fail, however, then we issue another test. If that test succeeds, we execute a different function and exit. But if the second test fails, then we run yet a third test.

In other words, we want to keep running tests until one succeeds. As soon as it does, then we execute some function(s) and exit. But as long as each test fails we will run another test until we reach some point where we give up. This construct is implemented by a `case` statement in other computer languages.

An example where this occurs in the AutoLISP language itself is in the `eval` function. Recall where `eval` tests for special operators. It first tests for `quote`, then `setq`, then `defun`, and so forth. If the operator is indeed `quote`, it runs `quote` and exits the test sequence. But if the operator is not `quote`, then it goes on to check each special operator in turn until it finds a match (or gives up if no match can be found).

As the above pseudo code shows, we can perform this task in AutoLISP with `if`. However, due to our indentation conventions, we quickly find our code beginning near the right-hand margin and wrapping around the screen. `cond` is really the more elegant way of handling this situation.

`cond` list . . . list	Evaluates the car of each list until one returns non-`nil`. It then evaluates all other forms in that list and returns whatever the last form evaluated returns.

The format of `cond` is as follows:

```
(cond (test body)
      (test body)
          .
          .
          .
      (test body))
```

Here's how `cond` works:

- `cond` takes as many arguments as we wish to give it; each argument must be a *list*.
- The car of each list is a test. The test forms are evaluated in order until one returns non-`nil`.

- When a given test returns non-nil, its body forms are evaluated in order and cond returns whatever the last form evaluated in that body returns. It does not attempt to evaluate any subsequent test.

- If all of the tests return nil, cond simply exits and returns nil.

If we compare these steps with the preceding if code segment, we can see that cond really performs the exact same actions as if. It's just a little bit more compact and vertically structured. cond's syntax, however, presents one confusing drawback, as the following example shows:

```
(cond ((equal (type data) 'str) (getstring))
      ((equal (type data) 'int) (getint))
      ((equal (type data) 'real) (getreal))
      (t (prompt "None of the above")))
```

On most lines there are two open parentheses. The one on the left is required by the syntax of the cond, because each of cond's operands is a *list*. Its matching right parenthesis is at the end of the line. The second open parenthesis is the normal parenthesis that begins an AutoLISP function call.

This is the only place in all of AutoLISP where we find code with two consecutive open parentheses. cond is a very old function that uses parentheses to group expressions together. On the plus side, since it does use parentheses to group, when a test succeeds we can have as many forms as we want evaluated without the need for progn. In this respect cond is easier to use than if.

The test on the first line is

```
(equal (type data) 'str)
```

This test asks whether the supplied datum is of type *string*. If it is, AutoLISP performs a getstring and exits the cond. If it's not a string, AutoLISP then tests whether the datum is an integer or real. If either of these tests succeeds, AutoLISP runs getint or getreal, respectively, then exits the cond. If all three tests fail, AutoLISP evaluates the fourth test, t. What does t mean here, and why is it not inside parentheses?

Any AutoLISP object can be a form (i.e., an argument to eval). Most often we supply a list because we want to run a function, but eval can handle any datatype. In this case, the form is simply the symbol t. Since it is a symbol we want evaluated rather than the name of a function, we do not put parentheses around it.

When `eval` is given a symbol, it returns the symbol's value. Since the value of the symbol `t` is always `t`, it is the symbol we use when we specifically want to indicate non-`nil`. In short, `t` is a test that always succeeds (this is a situation where we can get into trouble if we've changed the value of `t`, especially if we've set it to `nil`!).

The fourth line says, "If all previous tests have failed, then this one must succeed. Do what follows." If the `cond` gets to this point, it prints a message telling the user that nothing was chosen. This is a "safety net" that we use to prevent the `cond` from simply falling out the bottom and returning `nil`. It isn't required, but it saves us headaches when we're *absolutely certain* that one of our tests will succeed, but none does.

Conceptually `cond` is straightforward; it does the same thing as `if`. It may take some practice to get used to its awkward syntax but, once we do, it isn't really any harder than `if`.

6.3 Iteration Functions: Writing Loops to Repeat Code

Iteration means repetition; the functions in this section enable us to write loops. A *loop* is a block of code that runs repeatedly either a specified number of times or until some end condition is met. (Looping concepts are discussed in Section 14.3.2.)

A loop is an essential structure that is present in every computer language. It is beneficial, say, when we write an AutoCAD command to size a door and put it into a doorway, but we don't want to run this command for every single one of the 50 doors in our building. Instead, we write a loop that finds each door and places it into the doorway for us. Programming power comes through repetition, and a loop is the most efficient way of repeating tasks.

AutoLISP has three special operators for creating loops. Each possesses a specific capability, so we not only need to realize when a loop is needed, we must also decide which iteration function is best for our particular task.

- `repeat` is used when we know exactly how many times we need to repeat the loop.
- `foreach` is used when we want to perform one or more operations on each element in a list.
- `while` is used in all other looping situations.

`repeat` int body	Evaluates its body forms int times and returns whatever the last form evaluated in its body returns.

`repeat` provides a convenient facility for iterating over a block of code a specific number of times. Its syntax is as follows:

```
(repeat countform
  body)
```

When we run `repeat`, the *countform* is evaluated once prior to the first pass through the loop. It returns a positive integer that indicates the number of times to evaluate the body forms.

The *body* is a group of one or more forms that get evaluated in order, countform number of times. `repeat` ultimately returns whatever the last form in its body returns on the last pass through the loop.

```
Command: (setq counter 8)
8

Command: (repeat 5
1> (print counter)
1> (setq counter (1- counter)))
8
7
6
5
4 3
```

In this example the countform is 5, which indicates that the loop is to be executed five times. Prior to entering the loop, the variable `counter` is set to 8. The first expression in the body of the loop prints `counter`'s value, 8, on the screen. The second form then sets `counter`'s value to 7 by subtracting 1 from its current value (remember that `1-` is an AutoLISP subr). We call this *decrementing* the value of `counter`. Thus, `counter`'s value is 7 following the first pass through the loop.

On the second time through the loop, `counter`'s value, 7, is printed, then `counter` is decremented again. Thus, its value is 6 after this pass.

The loop is executed a total of five times. On the final pass, `counter`'s value, 4, is printed. The last form evaluated in the body of the loop sets `counter` to 3, so this is what `repeat` returns. The result is returned to the screen solely be-

cause we typed this code to the read-eval-print loop at the Command: prompt. If this code were contained within a `defun`, we would not see the 3.

We use `repeat` when we know exactly how many times we need to repeat a loop. For example, suppose we want to access each character in the string that is the value of the symbol `str`. We can sequence through the string with `repeat`:

```
(repeat (strlen str)
    .
    .
    .)
```

`strlen` is a subr that returns the number of characters in a string. Another subr, `sslength`, returns the number of entities in a selection set. We can use `repeat` with `sslength` to sequence through a selection set. These situations don't arise quite as often as we might like, so `repeat` is not that frequently used. In most situations `while` is the iteration operator that suits our needs.

`while` test body	Repeatedly evaluates the test and body forms until the test returns `nil`, then returns whatever the last form evaluated in the body returns.

The `while` function iterates over a block of code as long as some condition is met. It has the following syntax:

(`while` testform
  body)

The *testform* is evaluated prior to each pass through the loop. If it returns `nil`, the `while` is exited. If it returns anything but `nil`, all of the forms in the body of the `while` are evaluated. The *body* is a group of one or more forms that get evaluated in order. `while` ultimately returns whatever the last form in its body returns on the last pass through the loop.

```
Command: (setq c 8)
8

Command: (while (> c 3)
1> (setq c (1- (print c))))
8
7
6
5
4 3
```

This loop does the exact same thing as the example for `repeat` shown previously.

Prior to entering the loop, the variable `c` is bound to 8. Since its value is greater than 3, the body of the `while` is evaluated. The body consists of only one form, which actually has functions nested three levels deep. When functions are nested, the innermost parentheses are *always* evaluated first.

The innermost function, `print`, prints the value of `c`, 8, on the screen. It returns that value back to the `1-` function (recall that `1-` is an AutoLISP subr), which decrements the value of `c`. Thus `c`'s value is 7 following the first pass through the loop.

On the second time through the loop, `c`'s value, 7, is printed, then `c` is decremented again. Thus, its value is 6 after this pass.

The loop is executed a total of five times. On the final pass, `c`'s value, 4, is printed. The last form evaluated in the body of the loop sets `c` to 3, so the test, `(> c 3)`, fails, and `while` exits and returns 3. The result is returned to the screen solely because we typed this code to the read-eval-print loop at the Command: prompt. If this code were contained within a `defun`, we would not see the 3.

One question new users frequently ask is, "The last form evaluated in the `while` was the test, which returned `nil`. Therefore, shouldn't `while` return `nil`?"

Since `while` exits when the test returns `nil`, `while` will *always* have `nil` as its very last result. Having this returned is of no use to us. Thus, like most other functions that have a body, `while` returns the result of the last form evaluated *in its body.*

We must be careful to give `while` a test that we are certain will eventually fail. The following example shows a test that always succeeds, thereby creating an infinite loop:

```
(setq x 1)
(while (> x 0)
    .
    .
    .
  (setq x (1+ x)))
```

Since `x`'s value is initially positive and we add one to it every time through the loop, it's always going to be greater than zero and the loop will never end. In

this case the user must press <esc> to break out of the loop. When function calls are involved, the user might get an "AutoLISP stack overflow" error.

We can create an `until` function by applying `not` to the test of a `while`:

```
(while (not (zerop x))
       .
       .
       .)
```

The body of this loop is evaluated *until* the value of `x` becomes 0.

`while` is the most general iteration operator and certainly the most useful. It is an important function to add to your toolbox as soon as you can.

Example

The `c:long-dist` command measures the distances between points entered by the user. It repeatedly prompts the user to enter points and prints the distance between them. At the conclusion, it prints the total distance traversed and the average distance per leg. (`c:long-dist` is based on a command by Bill Kramer.)

Before studying this program, it's well worth running it first on our own machines. When the command prompts for a point, we can use our cursor to click on the screen. However, to make our example easier to follow, we'll type in easy numbers at the prompt instead. Here's what appears on our screen when we do:

```
Command: long-dist

Starting point (Enter when done): 0,0
Next point: 3,0      Distance: 3.0
Next point: 3,6      Distance: 6.0
Next point: 12,6     Distance: 9.0
Next point:

Total traversed distance: 18.0   Average: 6.0
```

As we can see when we run this command, it repeatedly prompts the user for points, then prints out the distance between the previous point and the current point. It stops looping when the user presses the <enter> key, then prints the sum of all the distances and the average distance per leg.

This function can be used to measure existing structures, such as the perimeter of a polygon. All of the standard AutoCAD object snaps work, so we can easily measure from endpoints or intersections. Now let's examine the code itself:

```
1(defun c:long-dist (/ tot-dist cnt p1 p2)
2   (initget 1)
3   (setq tot-dist 0.0                                  ;Initialize variables
          cnt      0                                    ;and get first point
          p1       (getpoint "\nStarting point (ENTER when done): ")
    )
4   (while (setq p2 (getpoint p1 "\nNext point: "))  ;Next point entry
5      (princ " Distance: ")                            ;Display distance during calc.
6      (setq tot-dist (+ tot-dist (prin1 (distance p1 p2)))
            cnt      (1+ cnt)                           ;Increment leg counter.
            p1       p2)                                ;Set new base point
    )
7   (and (> cnt 0)
         (princ "\nTotal traversed distance: ")
         (prin1 tot-dist)                               ;Total distance traveled.
         (princ " Average: ")
         (prin1 (/ tot-dist cnt)))
    (prin1))
```

1. `c:long-dist` is a user-defined AutoCAD command, not an AutoLISP function. Thus, it cannot have parameters, but it does create four local variables:

   `tot-dist` Records the total distance traversed. It's initialized to 0.0 (to ensure a real division), then each time through the loop, the new distance is added on. At the end of the function the value of `tot-dist` is printed.

   `cnt` Counter of the number of legs traveled. It's initialized to 0 and incremented each time through the loop. At the end of the function, `tot-dist` is divided by `cnt` to calculate and print the average.

   `p1` Contains the base point for each distance calculation.

   `p2` Contains the second point for each distance calculation.

2. This command, and others in this book, make liberal use of the powerful AutoLISP I/O functions. The use of `initget` here is subtle and not necessary to understand at this time.

3. Following the `initget` is a `setq` that initializes the variables `tot-dist`, `cnt`, and `p1` (remember, we can set the values of multiple variables with a single `setq`).

4. The core of this command is the `while` loop. The test to `while` is a `getpoint` function that prompts the user to type in or click on a point,

much as the TEXT command does when it requests a starting point for the text. When the user enters the point, it is returned by `getpoint` and assigned to the variable `p2`.

5. `princ` prints the string `" Distance:"` on the screen.

6. The only other function in the body of the `while` is `setq`, which changes the values of the same three variables we initialized.

   - `tot-dist` is set to the result of a fairly complicated computation. Within the expression that begins with `+`, the current value of `tot-dist` is obtained, then the second operand is evaluated. As always, the innermost parentheses are evaluated first. `distance` is a subr that takes two points as arguments and returns the distance between them. This result is returned to the `prin1` function, which prints the distance on the screen. `prin1` returns that same result back to the `+` function, which adds the distance to the current value of `tot-dist`. Finally, the new sum is assigned to `tot-dist` by the `setq`.

     In other words, this one line of code calculates and prints the distance from `p1` to `p2`, then adds this distance to the current value of `tot-dist`.

     This is a lot to accomplish with one `setq`, but it is standard AutoLISP style to nest forms many levels deep, evaluate the innermost first, then propagate the results back to the outermost level. If you're just starting to learn AutoLISP, you may find it easier to write each of these steps on its own line, with its own `setq`. As you gain experience you'll find it more economical and simpler to nest expressions as we've done here.

   - `cnt` is incremented by one and `p1` is set to be the same as `p2`. The latter step is worth examining.

     In our example, the first time through the loop, `p1`'s value is the point 0,0. A rubber-band line is drawn on the screen until the user selects the point 3,0. This point is then assigned to `p2`.

     Within the loop, `p1` is always the base point, and `p2` is the new point. Thus, at the end of the loop we must bump `p1` to be the same as `p2` so that the former new point can become the base point the next time through the loop.

   Since the `setq` is the last form in the body of the loop, the loop is now done. It goes back to the top, gets the next point, and repeats the body again. In the sample execution of `c:long-dist` the loop is repeated three times. What makes it terminate?

The while loop exiting depends on a trick inherent in the getpoint function (and most other get functions as well). getpoint requires that the user type in or click on a point. However, instead of entering a point, the user has the option of pressing the <enter> key. When that happens, getpoint returns nil instead of a point.

At the start of c:long-dist the user is instructed to hit <enter> when done. When the user presses <enter>, getpoint returns nil, which gets assigned to p2 via the setq. More important, setq returns the nil as a result, and it becomes the first argument to while. When while is given nil as the result of the test, it automatically exits.

This method of exiting the while is actually quite profound. Rather than explicitly running a test via some predicate, we run a function that performs meaningful work and returns nil when it has completed. This makes our code more compact and efficient. There are several functions we'll cover later in this book that return nil when they have completed running. These functions are excellent candidates for use as a test to the while.

7. When the user hits <enter> to indicate loop termination, c:long-dist prints the total distance and average. But one important step must be done first. Suppose the user presses <enter> on the first pass through the loop. At that time the value of cnt is still 0, and if the command divides tot-dist by cnt, it will get a "divide by zero" error. In any computer language it is illegal to divide by zero because that's a mathematical impossibility. Therefore, prior to dividing, it must test whether cnt is 0.

   We could have used if to accomplish this task. The code segment would then appear like this:

```
(if (> cnt 0)
    (progn
      (princ "\nTotal traversed distance: ")
      (prin1 tot-dist)                          ;Total distance traveled.
      (princ " Average: ")
      (prin1 (/ tot-dist cnt))))
```

   Instead, we chose to use and in order to show how this subr would stand alone, *not* inside of an if. This is a subtle nicety that is an example of elegant programming, but it's not very important.

   Recall that and evaluates its forms in order as long as they return non-nil. Within this and, the first form tests whether the value of cnt is greater than 0. If it's *not,* the test returns nil, the and exits, and c:long-dist is done.

On the other hand, if the value of `cnt` *is* a positive number, all of the other forms in the body of the `and` are evaluated. The other functions, `prin1` and `princ`, are variations of the `print` subr that will *never* return `nil` (unless we explicitly tell them to print `nil`). Therefore, > is the only true test in the body of the `and`, and the only function that could possibly cause the `and` to exit. In this case, `and` accomplishes the same task as `if`, without the overhead of the `progn`: It prints the total distance and the average distance, along with their string headings.

`foreach` sym list body	Executes the body once for every member of the list. Each time through the loop, sym is bound to the next list element. It returns whatever the last form evaluated in its body returns.

A common AutoLISP task is to sequence through a list to perform one or more operations on each of its elements. When this situation arises, we can use the `foreach` special operator to iterate through the list. Its syntax is as follows:

```
(foreach sym listform
  body)
```

sym is a local variable that is used within the loop. It is identical in concept to the local variables that we specify following a forward slash in the parameter list of a `defun`, but in `foreach` it is local to the loop itself. The variable is created when the loop is entered and destroyed when the loop is completed. We can supply any valid symbol name for sym, although we typically give it a name that somehow indicates its purpose in the loop. Note that we *do not* define this variable in the function's parameter list.

The local variable is used to access the elements in the list. Each time through the loop, sym is automatically bound to the next member of the list.

Prior to the first pass through the loop, *listform* is evaluated and must return a list that the loop can access and "iterate over." The list may be supplied directly, as in the following example, or we can build a list composed of user input. The loop is executed once for each list element. Thus, a 10-element list will cause the loop to be executed 10 times, a 50-element list will cause the loop to be executed 50 times, and so on.

The *body* is a group of one or more forms that get evaluated in order. `foreach` ultimately returns whatever the last form in its body returns on the last pass through the loop. Consider this example:

```
Command: (foreach x '(a b c)
1> (print x))
A
B
C C
```

Since the list, `(a b c)`, has three elements, the loop is executed three times. The first time through the loop, the value of `x` is `a`, the first element in the list. `(print x)` causes the symbol `a` to be printed on the screen.

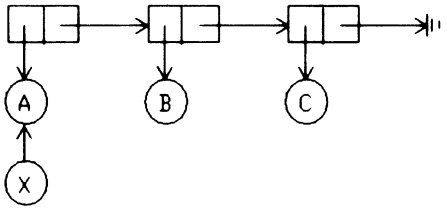


Figure 6-4

When the body forms have all been evaluated (in this case there's only one form), `foreach` automatically goes back to the start and rebinds `x` to have the second element of the list as its value:

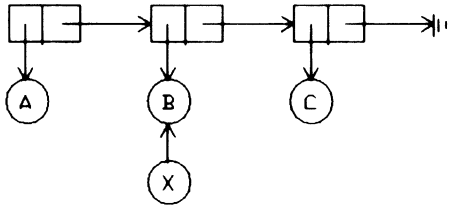


Figure 6-5

Thus, on the second pass, the symbol `b` is printed on the screen. Then, on the third and final pass, the symbol `c` is printed:

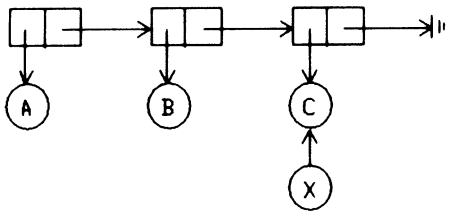


Figure 6-6

At this point `foreach` knows that the list has been exhausted, and it automatically exits and returns the last form evaluated in its body, `c`. `c` is printed on the screen a second time solely because we typed this code to the read-eval-print loop at the Command: prompt. We would normally run this loop inside of a `defun`, and `c` would be printed only once.

Example 1

The `circ` function takes a list of points and draws circles of radius 0.2 centered at each point. We would run it as follows:

```
Command: (circ '((2 4) (5 6)))
nil
```

This call to the `circ` function draws circles whose center points are 2,4 and 5,6. Let's examine the `circ` function itself:

```
1  (defun circ (plist)
2    (foreach cen plist
3      (command "circle" cen 0.2)))
```

1. The `circ` function has one parameter, `plist`, which gets as its value the list of points that is supplied in the function call. Since a 2D point is itself a list of two numbers, `plist`'s value is a *list of lists:*

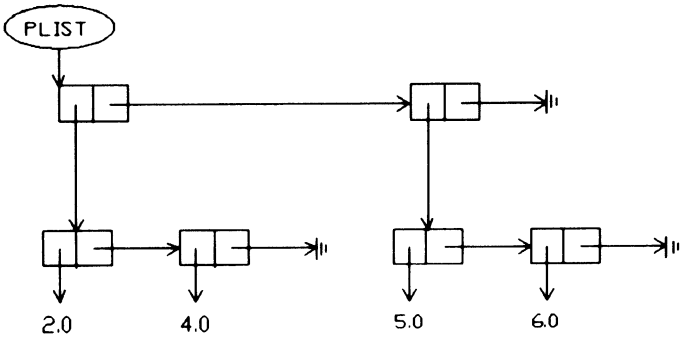


Figure 6-7

2. When the `foreach` is entered, its local variable `cen` is bound to the first element in this list:

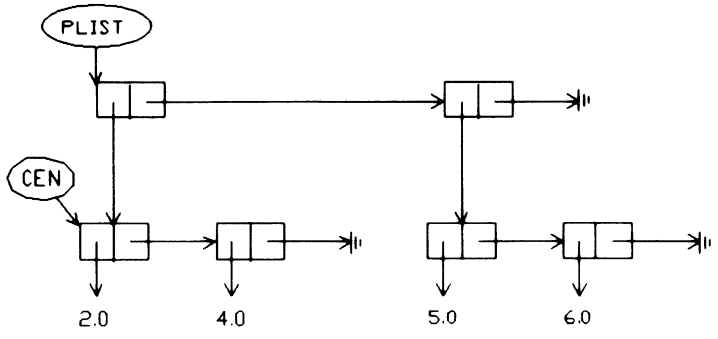


Figure 6-8

3. The body of the `foreach` consists of one form, a `command` function. This function has the AutoCAD CIRCLE command draw a circle of radius 0.2 centered at 2,4, the value of the variable `cen`.

The first pass through the loop is finished, so the `foreach` goes back to the start and rebinds `cen` to have the second point as its value:

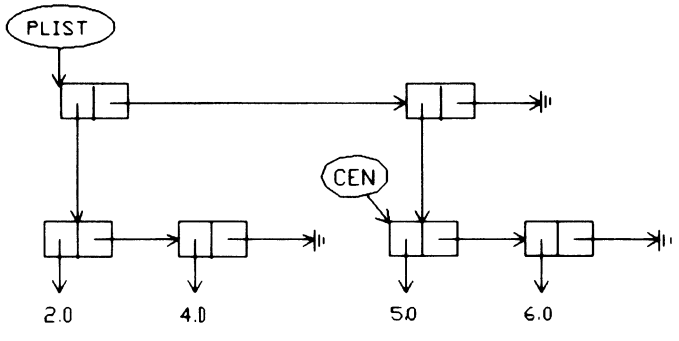


Figure 6-9

The loop repeats, this time drawing a circle centered at 5,6. When finished, the list is completed, so the `foreach` exits and the function terminates.

Incidentally, if we replace 0.2 with `pause` in the `defun`, the function pauses on each pass and allows the user to specify the radius of the point. Once we have written a function in AutoLISP, we frequently discover simple modifications that accomplish other useful tasks. Be inventive!

Example 2

The following function, bounds, is a more elaborate variation of the cen function. It also takes a list of points as an argument, but its task is to determine which points, if any, are out of bounds. It prints a message on the screen for each point that is beyond the specified limit.

bounds only tests the X coordinate of each point. The point is deemed to be out of bounds if its X coordinate is greater than 10. This bound was arbitrarily chosen to keep the example relatively simple. We could easily modify this function to test a different bound (such as the X drawing limit), and we could add a test for the Y and/or Z coordinate as well.

Prior to running the bounds function we assign a list of points to the variable points:

```
Command: (setq points '((3.98777 4.90979) (10.0372 6.204) (2.1162 11.306) (12.429 8.44657)))
((3.98777 4.90979) (10.0372 6.204) (2.1162 11.306) (12.429 8.44657))
```

This list of points appears as follows in our picture notation:

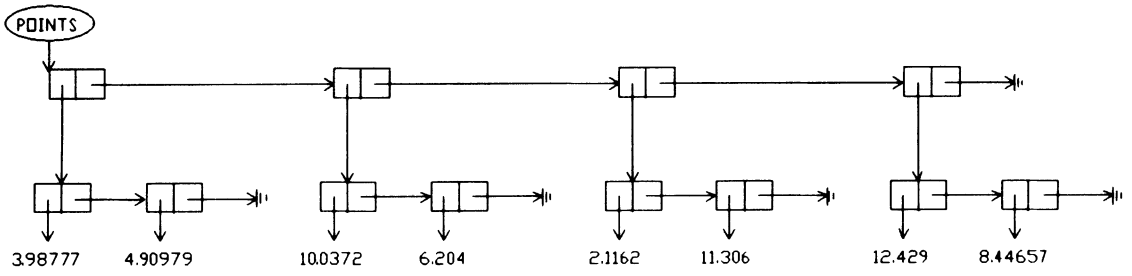


Figure 6-10

As we can see, the second and fourth points have X coordinates greater than 10. These are the points that are flagged when we run the bounds function:

```
Command: (bounds points)
The point 10.0372,6.204 is out of bounds
The point 12.429,8.44657 is out of bounds
```

Now let's examine the code itself:

```
1 (defun bounds (pt-list)
2   (foreach pt pt-list
3     (if (> (car pt) 10)
4         (prompt (strcat "\nThe point " (rtos (car pt)) "," (rtos (cadr pt))
                          " is out of bounds"))))
5   (prin1))
```

1. `bounds` is a function that takes one argument, and therefore must have one parameter. When we type `(bounds points)` to run the function, `points` is evaluated and its value, the list of points, is passed to the function as the argument. This list is then bound to the parameter, `pt-list`, giving the following relationships:

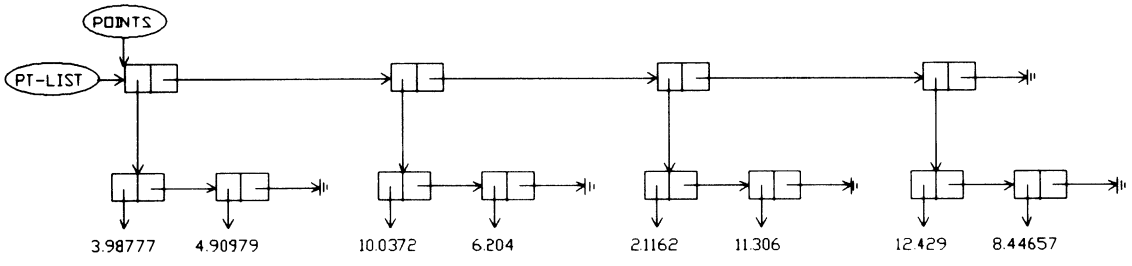


Figure 6-11

This is a setup we've encountered before: Outside the function the list is the value of the variable `points`; inside the function it is the value of `pt-list`. It doesn't matter what we call the list because each symbol is ultimately evaluated and the list itself is used.

2. The entire body of the `bounds` function is a `foreach` loop. In the loop, `pt` is the local variable that will access each element in turn. In other words, `pt`'s value will be the next point in the list each time through the loop. The list itself is accessed via the variable `pt-list`.

3. The body of the `foreach` consists of an `if` function. The test to the `if` asks, "Is the car of `pt` greater than 10?"

   The first time through the loop, the value of `pt` is the first element in the list, that is, the point `(3.98777 4.90979)`:

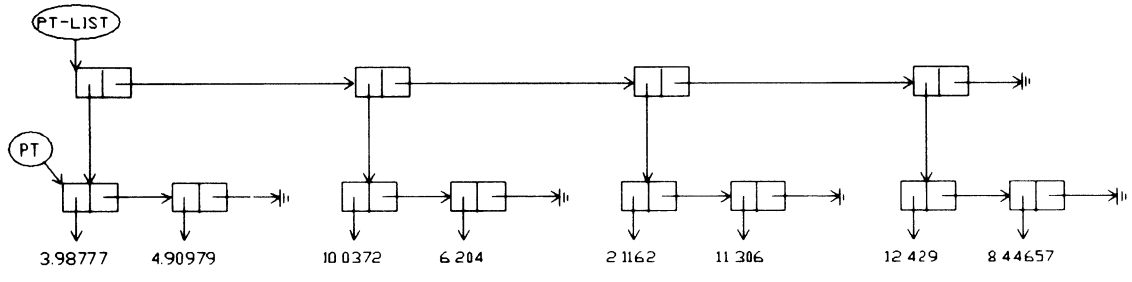


Figure 6-12

The car of this list is the X coordinate, 3.98777. Since this is not greater than 10, the test fails.

The `if` in this example is an IF-THEN, rather than an IF-THEN-ELSE expression. When the test fails, there is nothing more to do (remember, we only want to flag those points whose X coordinates are greater than 10), so the `if` just exits and returns `nil`. Since the `if` is the only form in the `foreach`, the first pass through the loop is completed.

The `foreach` automatically goes back to the start of the loop and bumps `pt` to have the second point, `(10.0372 6.204)`, as its value:

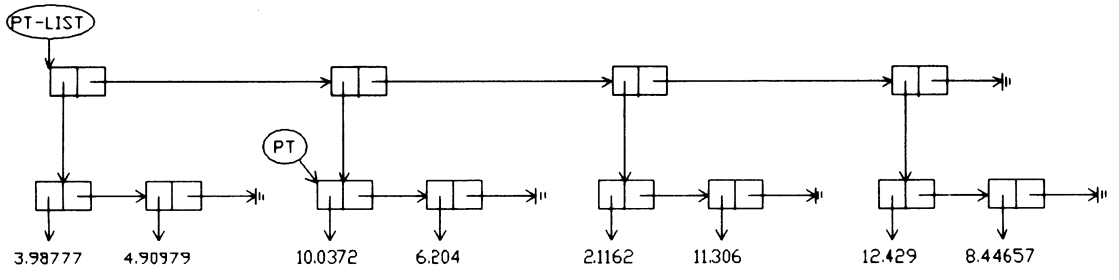


Figure 6-13

The second time through the loop `if`'s test succeeds because the car of `pt`, 10.0372, is greater than 10. This time `if` takes the THEN path.

4. The THEN path consists of a single form that contains three subrs we've not yet seen:

   `prompt` A variation of `print` that outputs a string to the user's screen.

   `strcat` Takes strings as arguments and joins them together into one long string.

   `rtos` Takes a real number as an argument and converts it into a string.

   Since the code in the innermost parentheses is evaluated first, let's start there. `(car pt)` returns the X coordinate of the point, 10.0372; `(cadr pt)` returns its Y coordinate, 6.204. Each of these points is converted from a real number into a string by the `rtos` function.

   `strcat` next joins five string segments

   ```
   "\nThe point" "10.0372" "," "6.204" "is out of bounds"
   ```

   into one long string. This string is handed in turn to `prompt`, which prints it on the screen.

Once the string has been printed, the `if` is done, as is the second pass through the loop. The loop is executed twice more. Nothing is printed for the point `(2.1162 11.306)` because its X coordinate is less than 10. The X coordinate of the last point, `(12.429 8.44657)`, *is* greater than 10, so a second message is printed on the screen.

5. Following the fourth pass through the loop there are no more points to examine, so the `foreach` automatically exits. The last form in the `bounds` function, `(prin1)`, is used for formatting.

6.4 Advanced Topic: Writing Recursive Functions

A *recursive* function is a function that calls itself. We saw an example of recursion when we studied `eval`. The AutoLISP language is structured in a way that enables us to write our own recursive functions using `if` or `cond`. (Recursion concepts are explained in Section 14.3.2.)

In some situations recursion allows for very elegant solutions; in others recursion is necessary. But most of the time we can solve our problems using a simple `while` loop instead of the somewhat trickier recursive method. Therefore, recursion is a topic that is best left alone until we are comfortable using AutoLISP in general.

When writing recursive functions, we generally include the following steps:

- Examine the argument by testing for the simplest case. The test should ask, "Are we done?"
- If the test succeeds (i.e., returns non-`nil`), then the recursion is finished. Return a simple result.
- If the test fails (i.e., returns `nil`), call the function again with an argument that has been simplified.

Recursion is best understood by looking at examples.

Example 1

Let's write a recursive function that computes the number of elements in a list. We might run this function as follows:

```
Command: (len nil)
0
```

```
Command: (len '(a))
1

Command: (len '(a b))
2
```

Here's the recursive code that accomplishes this task:

```
(defun len (listx)
   (if (null listx)
       0
       (+ 1 (len (cdr listx)))))
```

The `len` function takes one argument, a list, which becomes the value of the parameter, `listx`.

The entire body of `len` is an `if` statement. `if` examines the list to see whether it is a null list. In other words, it tests whether it has the symbol `nil`. If the value of `listx` is indeed `nil` (as it is when we enter `(len nil)`), the end of the list has been reached:

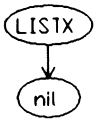


Figure 6-14

The THEN path is taken and the number 0 is returned by the `if` and by the `len` function as well. This is what happened with our first call to `len`.

If the value of `listx` is not `nil`, the ELSE path is taken. 1 is added to the result of calling the `len` function again, this time with the *cdr of the list.* In other words, "first-level" `len` calls "second-level" `len` and uses the result that is returned as the second argument to the + function. This is *exactly* analogous to the way `eval` works.

Let's see what happens when we pass `len` the one-element list, `(a)`:

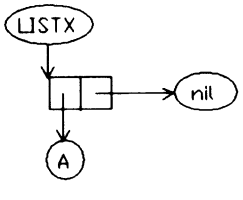


Figure 6-15

Since the value of `listx` is not `nil`, the ELSE path is taken. 1 is added to what `len` returns when passed the cdr of the list:

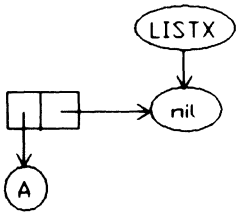


Figure 6-16

In this case, the cdr of the list is `nil`. Thus, "second-level" `len` takes the THEN path and returns 0. The arguments 1 and 0 are added, and the result, 1, is returned by `len`.

Finally, let's see what the `len` function does with the list `(a b)`:

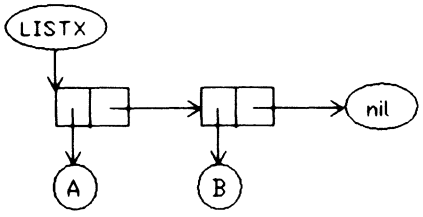


Figure 6-17

"First-level" `len` is supplied the list `(a b)`. Since this isn't `nil`, it adds 1 to the result of calling "second-level" `len` with the argument `(b)`:

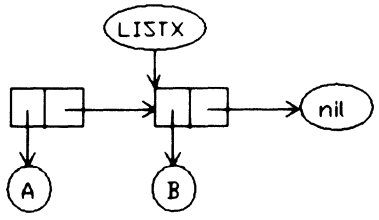


Figure 6-18

"Second-level" sees that it doesn't have `nil` either, so it adds 1 to the result of calling "third-level" `len` with the argument `nil`:

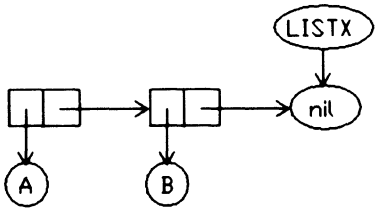


Figure 6-19

"Third-level" does have `nil`, so the `if` test succeeds, and "third-level" returns 0 back to "second-level."

"Second-level" adds the result, 0, to its first argument, 1, and returns 1 to "first-level."

"First-level" adds the result, 1, to its first argument, 1, and returns 2 to us as the final result of the `len` function.

Example 2

`find-last` returns the last element in a list. We would run it as follows:

```
Command: (find-last '(a b c))
c
```

Here's the recursive code to accomplish this task:

```
(defun find-last (elts)
  (if (null (cdr elts))
      (car elts)
      (find-last (cdr elts))))
```

`find-last` is similar to the previous example but has some notable differences:

- Like the `len` function, `find-last` "cdrs" down a list until it encounters `nil`. `find-last`'s test, however, is whether the cons' *cdr* is `nil`; to return the last element, it must have the *last cons* in its hand. If it gets to `nil` (like the `len` function does), it's gone too far and cannot backtrace to get the last element.

- Once it finds the last element, it doesn't perform any operation on it; it simply returns that element.

Here's how `find-last` handles the list `(a b c)`:

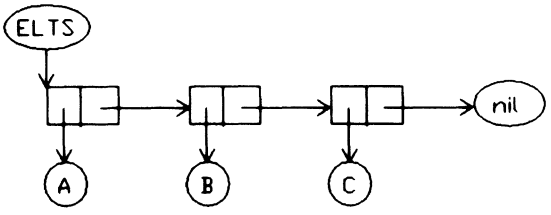


Figure 6-20

The list is the value of the parameter `elts`. "First-level" `find-last` checks whether `(cdr elts)` is `nil`. In this case the cdr isn't `nil`, so it calls "second-level," passing it the cdr of the list, `(b c)`.

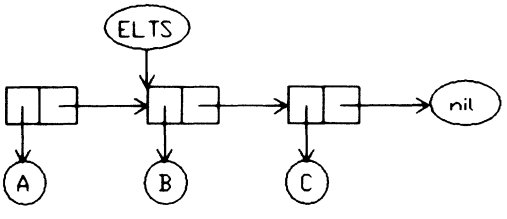


Figure 6-21

"Second-level" `find-last` sees that the cdr of its list is also non-`nil`. So it calls "third-level," passing it the cdr of the list, `(c)`.

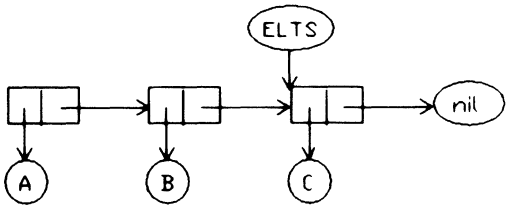


Figure 6-22

"Third-level" `find-last` discovers that the cdr of its list is `nil`, so it knows it's done. It takes the THEN path of the `if` and returns `(car elts)`, which is the symbol `c`, to "second-level."

"Second-level" receives the result of its call to `find-last`, so it is finished running. It simply returns the result, `c`, back to "first-level."

"First-level" `find-last` is also done, so it returns the same result, `c`, back to us.

Note: We can write this function iteratively:

```
(defun f-l (listx)
  (foreach memb listx
    elt))
```

The body of the `foreach` simply evaluates the symbol `elt`. The last time through the loop, `foreach` returns what the last form evaluated returns; that is, the last element of the list.

Recursive solutions may be elegant, but they're not always the easiest approach to solving a problem. In fact, these first two examples replicate the tasks of the `length` and `last` subrs, which are shown in Section 7.1.2.

Example 3

Another common recursive example that is very useful for some mathematical applications is the computation of N-factorial. This function takes a positive integer as its only argument and computes the product of multiplying itself by itself minus one, itself minus 2, and so on, down to 1. For example,

5-factorial = 5 * 4 * 3 * 2 * 1

or

5-factorial = 5 * 4-factorial

We would run this function as follows:

```
Command: (factorial 3)
6

Command: (factorial 5)
120
```

Try to write N-factorial before looking at the code and the explanation. Note that unlike the previous recursive functions, this one works with integers, not lists. Also be aware that 0-factorial and 1-factorial are both 1.

```
(defun factorial (n)
  (if (= n 1)
      1
      (* n (factorial (1- n)))))
```

Here's what the `factorial` function does:

Since `factorial` starts with a positive integer and subtracts 1 each time, the logical end test is to check whether the argument to the function is the num-

ber 1. If it is, the function simply returns 1-factorial, which is 1. Note that we could test for 0 as well.

If the argument to the `factorial` function is not 1, it multiplies the argument by the result of running `factorial` again, this time with one less than the number it currently has.

This is how the `factorial` function handles the number 3:

"First-level" `factorial` is supplied the number 3, which becomes the value of the parameter, `n`. Since `n`'s value is not 1, "first-level" multiplies the value of `n` by the result of calling itself with the number 2.

"Second-level" sees that its argument, 2, is not 1, so it multiplies 2 by the result of calling itself with the number 1.

"Third-level" finds that its argument *is* 1, so it simply returns the 1 to "second-level."

"Second-level" now multiplies its two arguments, 2 and 1, and returns the result to "first-level."

"First-level" multiplies its two arguments, 3 and 2, and returns the result, 6, to us.

Notes

- Sometimes function `f1` calls `f2`, which calls `f3`, which in turn calls `f1`. This is another form of recursion.
- Recursion provides short, elegant solutions to some types of problems. However, there are generally other ways to solve the same problem. So don't worry if you don't really understand recursion; it's a difficult concept that takes time and practice to become comfortable with.

6.5 Important Points to Remember

- *Predicates* are used to ask yes/no or true/false questions. They are most frequently used as the test for an `if` or a `while`. It is customary to give our own predicates names that end in the letter *p*.
- Our signal to use a *conditional* function comes when we say to ourselves, "If this happens, then I want to perform that task."

- `cond` and `if` are completely interchangeable. `cond` is especially helpful when we need to run multiple tests, then perform some actions when one of those tests succeeds.

- `and`, `or`, and `not` usually modify a test to an `if` or a `while`. `and` and `or` can stand alone, however.

- When we need to perform a group of tasks many times, then one of the iteration functions is called for. If we know exactly how many times we want to perform the steps in the loop (even if that's a number that the user will enter), then we use the `repeat` function. If we want to access each element of a list, then `foreach` is usually the function we want. `while` is the correct tool for all other looping situations.

- A limitation of the looping functions is that we have no mechanism for ending a loop prematurely. This impacts us most severely with `foreach`, because we frequently want to exit the loop when we encounter a specific element in the supplied list. When this happens it is sometimes necessary to use `while` even though we are iterating over lists.

6.6 Labs

Note: Some of the labs in this chapter (and subsequent ones) draw or change something on the screen rather than manipulate data, and I can't easily present examples of how they are run. For example, the upcoming Lab Exercise 7 asks you to write a command to ZOOM PREVIOUS. If you don't understand what is requested for one of these exercises, load the `answers.lsp` file and run the function or command. Once you've seen it run your task will be apparent.

Predicates and Conditionals

1. This exercise should be done at the Command: prompt. Initialize the environment by entering the following forms:

```
Command: (setq x 1 y 2 z 3)

Command: (setq a (list x y z))

Command: (setq b '(1 2 3))

Command: (setq c (list 'x 'y 'z))

Command: (setq d b)
```

What do the following functions return? Write down your answers, then test them on the machine.

```
Command: (numberp 5.0)

Command: (zerop 5.0)

Command: (minusp 5.0)

Command: (type 5.0)

Command: (type b)

Command: (type (car a))

Command: (type (car b))

Command: (type (car c))

Command: (eq a b)

Command: (eq b d)

Command: (equal a b)

Command: (equal a c)

Command: (eq a a)

Command: (equal x (cadr b) 1.1)

Command: (eq "abc" "abc")

Command: (eq "abc" "ABC")
```

2. What is the difference between the `not` and `null` functions?

3. Write a predicate called `evenp` that takes a number and returns `t` if the number is even and `nil` if it is odd. Hint: The `rem` function is useful here.

```
Command: (evenp 2)
T

Command: (evenp -3)
nil
```

4. Write a predicate called `symbolp` that returns `t` if its argument is a symbol and `nil` otherwise. Hint: Use the `type` function.

```
Command: (symbolp 3)
nil

Command: (symbolp 's)
T
```

5. The `fix` function is called `floor` in Common Lisp because it converts its argument to an integer by truncating downward. Write a function called `ceiling` that works the same except that it converts by always rounding upward, away from zero.

```
Command: (ceiling 4.1)
5

Command: (ceiling -4.1)
-5

Command: (ceiling 4.0)
4
```

6. Write a function that uses a `cond` to analyze a grade:

   If the grade is 90 or higher, return `excellent`
   80–89 return `good`
   65–79 return `not-so-good`
   < 65 return `failing`

```
Command: (grade 93)
EXCELLENT

Command: (grade 68)
NOT-SO-GOOD
```

   When this function works, rewrite it using `if`. It will give you a feeling for the usefulness of `cond`.

7. Write a function named `zm` that takes the string `"w"`, `"p"`, or `"e"` (use a `cond` and test for both uppercase and lowercase strings) as an argument and performs a ZOOM WINDOW, PREVIOUS, or EXTENTS, respectively. You can base it on Chapter 5, Lab Exercise 8.

```
Command: (zm "e")                    ;The drawing zooms to its extents
nil
```

8. Write a function that takes two arguments. Verify that both arguments are numbers and that the second is between 2 and 4. If so, raise the first number to the power specified by the second. If not, return the symbol `error`.

```
Command: (expon 2 4)
16

Command: (expon 2 5)
ERROR

Command: (expon 2 "3")
ERROR
```

What would happen if we called this function `expt`?

Iteration and Recursion

Note: These functions can all be written iteratively. If the problem statement is followed by (rec), the Answers appendix contains a recursive solution as well.

9. Write a function that takes a number and a list and "cdrs" the list *number* times.

```
Command: (nthcdr 3 '(q r s t u v))
(T U V)

Command: (nthcdr 2 '(1 2 3 4 5 6 7 8))
(3 4 5 6 7 8)
```

10. Write a function that prints the third element of a list five times.

```
Command: (third-5 '(a b c d e))
C
C
C
C
C
```

11. Write a function that accepts a list of numbers and prints each number multiplied by 2.

```
Command: (multby2 '(1 3 6.2))
2
6
12.4
```

12. Write a function that prints each element of a list until a non-number is encountered. Return a count of the numbers printed prior to the first non-number.

```
Command: (non-num '(1 3 5 7 a 2 4 6 8))
1
3
5
7
"Numbers at the start of the list: " 4
```

13. Write a function that returns the sum of a list of numbers (rec).

```
Command: (sum (list 1 2 3 4 5))
15
```

14. Write a predicate that tests whether there are only numbers in a list (rec).

```
Command: (numsp '(2 4 6))
T

Command: (numsp '(a 4 6))
nil
```

15. Write a function that returns the largest of a list of numbers (rec).

```
Command: (largest '(3 8 4 7.3 6))
8
```

16. Write a function that returns the last cons in a list (rec).

```
Command: (last-cons '(a b c))
(c)
```

17. Write a function called `zi` that repeatedly prompts the user for a point (you will need `getpoint` for this exercise) and performs a ZOOM CENTER at that point. Make the size be 1/2 the amount returned by the VIEWSIZE system variable. This provides us with a progressive ZOOM IN on the drawing. Exit the loop when the user presses `<enter>`.

    When this function works, make a copy and call it `zo`. Alter this copy to ZOOM OUT by doubling the size specified by the VIEWSIZE system variable. Then add `"i"` and `"o"` options to the `cond` you created in Lab Exercise 7.

18. Write a function that accepts a list as its only argument and prints every other element of the list.

```
Command: (every-other '(a b c d e))
A
C
E
```

19. Write a function that takes a list of strings and a list of points. Place each string at the corresponding point in the drawing (you can base this on the `ins-str` function from Chapter 5, Lab Exercise 7).

```
Command: (put-strs '("abc" "def") '((3 3) (3 5)))
```

The function draws `"abc"` at 3,3 and `"def"` at 3,5.

20. Write a function to determine whether a given number is prime. A number is prime if it is only divisible by itself and 1.

```
Command: (primep 13)
T

Command: (primep 12)
nil
```

21. Write a function that prints all primes up to a given number.

```
Command: (prime-print 13)
1
2
3
5
7
11
13
```

7

Working with Lists

Topics Covered in This Chapter

- List functions
- Association lists
- Advanced topic: `apply`, `mapcar`, `lambda`

Goals for This Chapter

- Learn various functions that work specifically with lists.
- Solidify our knowledge of lists and sublists, and how to use the list functions to manipulate them.
- Learn the organization of association lists, which are used to implement the entity database.
- Learn the usage of several more-advanced list functions.
- Practice accessing and altering data in lists and association lists.

Introduction

Inasmuch as the name *LISP* is an acronym for ***LISt Processing***, it is not surprising that we constantly build and utilize lists. To this point we have seen how to create and access data lists and how to use lists that represent code.

In this chapter we'll learn about a dozen AutoLISP subrs that specifically manipulate data lists. They won't do everything we want, but whenever we need to run a function that does not exist, we can simply write it ourselves.

We'll also be introduced to association lists, which are the core of the entity database. Understanding these is essential for altering the entities in our drawings.

7.1 Basic List Functions

In this section we'll examine the various list subrs. But we'll approach them in a different way from other chapters.

Do you like challenges? If so, then try the quiz in the following section. If you want "just the facts, ma'am," then go directly to Section 7.1.2 where these functions are defined. Note that Section 7.1.2 contains numerous interesting and practical observations and is worth reading even if you know all of the quiz answers.

7.1.1 List Functions Quiz

Let's look at a list of some recent U.S. presidents, which is the value of the symbol `pres`:

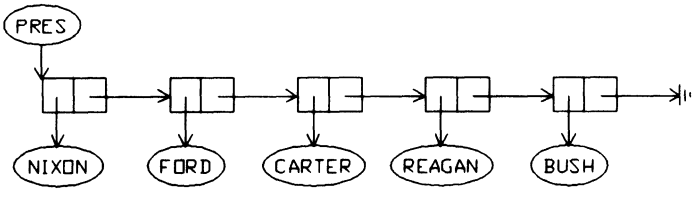


Figure 7-1

Try answering the following questions about this list. Even though we haven't yet seen some of these functions, we can often deduce their contracts from their names . . . you'll be surprised at how intuitive AutoLISP frequently is. Write your answers on a piece of paper, then check them in the following section. Each question uses the structure that was built in the previous step.

1. What would we enter to create this list (we've actually seen two very different methods for accomplishing this)?

2. What is the list's

   car
   cdr
   cadr

3. What would you guess is returned when we enter the following:

   ```
   Command: (length pres)

   Command: (last pres)

   Command: (nth 3 pres)
   ```

   `nth` returns the "nth" element of a list but may not do exactly what you think. We're not grading you, so give it a try!

   ```
   Command: (reverse pres)

   Command: (third pres)
   ```

4. We'd like to add `clinton` to the front of the list, like so:

   ```
   (CLINTON NIXON FORD CARTER REAGAN BUSH)
   ```

   How can we add `clinton` without building an entire new list?

   (Hint: It's a function we've already seen. What are we actually creating here?)

5. Now create a list of U.S. vice presidents and make it the value of the symbol `veeps`:

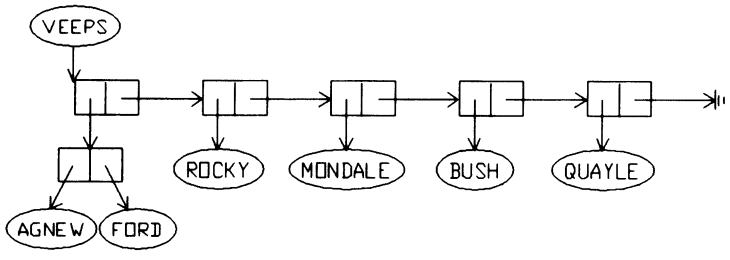


Figure 7-2

6. What do you suppose the following function does?

```
Command: (setq admin (append pres veeps))
```

7. We can go into these lists in various different ways and access `rocky`. Can you suggest two?

8. What do you think the following functions do? What do they return?

```
Command: (setq admin (subst 'reagan 'ford 'admin))

Command: (member 'rocky veeps)

Command: (member 'agnew veeps)
```

7.1.2 Quiz Answers

You'll understand these functions better if you type them in as you read this section.

1. What would we enter to create the list in Figure 7-3?

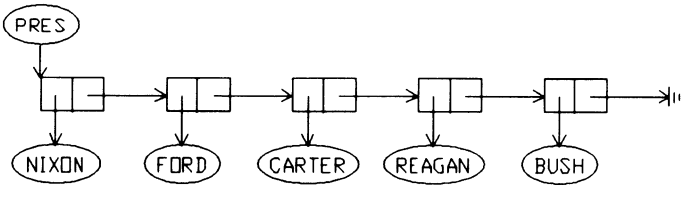


Figure 7-3

This list can be created in either of two ways:

```
Command: (setq pres (list 'nixon 'ford 'carter 'reagan 'bush))
(NIXON FORD CARTER REAGAN BUSH)

Command: (setq pres '(nixon ford carter reagan bush))
(NIXON FORD CARTER REAGAN BUSH)
```

The first example creates a list using the `list` function, then assigns that list to the variable `pres`. Each element in this list must be quoted or it will be evaluated:

```
Command: (setq pres (list nixon ford carter reagan bush))
(nil nil nil nil nil)
```

The second example quotes the typed-in list so that it's not evaluated and makes that list be the value of `pres`. When we type that list to the read-eval-print loop, the reader builds it and hands it to `eval`. By using `quote` we're saying, "This list is data, don't evaluate it, just return it as is." Thus `pres` is bound to the list itself.

These two examples do the exact same thing, so which is better? Most programmers prefer the second because it requires less typing. But in certain situations, quoting a list will not accomplish our intended purpose. Let's enter the following code:

```
Command: (setq x 3 y 4)
4

Command: (setq l (list 'x 'y (* x y)))
(X Y 12)
```

This example builds a list consisting of the symbols `x` and `y` and the product of multiplying their values. We quote the symbols `x` and `y` to prevent their evaluation, but we want the list `(* x y)` evaluated. If we quote the entire list, the multiplication form will never be evaluated:

```
Command: (setq l '(x y (* x y)))
(X Y (* X Y))
```

The usage rule is this:

> When we explicitly supply data to be collected in a list, we can type it in as a quoted list. If *any* of the operands needs to be evaluated, we use the `list` function to build the list.

2. Find out some information about this list.

```
Command: (car pres)
NIXON

Command: (cdr pres)
(FORD CARTER REAGAN BUSH)

Command: (cadr pres)
FORD
```

If you do not fully understand these answers, please reread Section 1.10; it is essential that you understand cars and cdrs before proceeding.

3. What do the following functions return?

`length` list	Returns the number of elements in a list.

```
Command: (length pres)
5
```

The length of a list is also the number of conses in its backbone. It's frequently easier to count backbone conses than it is to count elements. The following data list has only two elements: the large sublist and `h`:

```
Command: (length '((a (b c (d . e) f) g) h))
2                  └──────────────────┘ └┘
```

`last` list	Returns the last element in a list.

```
Command: (last pres)
BUSH
```

We don't use `last` to access the Y coordinate of a point; if the user supplied a 3D point it would return the Z coordinate instead. Use `cadr` to obtain the Y coordinate and `caddr` to get the Z.

nth n list	Takes an integer (represented by n) and list and returns the *nth* element of the supplied list, starting with position 0.

```
Command: (nth 3 pres)
REAGAN
```

At first glance we might suppose that this example would return the third element of the list, `carter`. However, `nth` is a peculiar function in that it begins counting from 0. Thus:

```
Command: (nth 0 pres)
NIXON

Command: (nth 4 pres)
BUSH

Command: (nth 5 pres)
nil
```

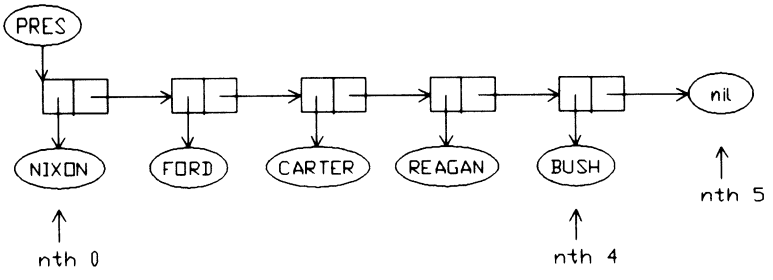


Figure 7-4

`nth` is derived from a substantial collection of Common Lisp functions that are zero-based. Curiously, all of the others are irrelevant to AutoCAD and have been removed from AutoLISP. On the other hand, Autodesk added a function for entity handling called `ssname` that works quite similarly to `nth`; it also begins counting from zero.

reverse list	Returns a new list with the elements of the supplied list in reverse order.

```
Command: (reverse pres)
(BUSH REAGAN CARTER FORD NIXON)

Command: !pres
(NIXON FORD CARTER REAGAN BUSH)
```

reverse is the first of several AutoLISP functions we'll see that appear to alter an existing list but actually create a modified copy of the original structure. Although the reversed list is printed on the screen (if the code is entered at the Command: prompt), the original list remains unchanged. If we wish to save the new list, we must setq it to a variable:

```
Command: (setq p (reverse pres))
(BUSH REAGAN CARTER FORD NIXON)
```

Now the new list is the value of the variable p, while the original list is still the value of pres:

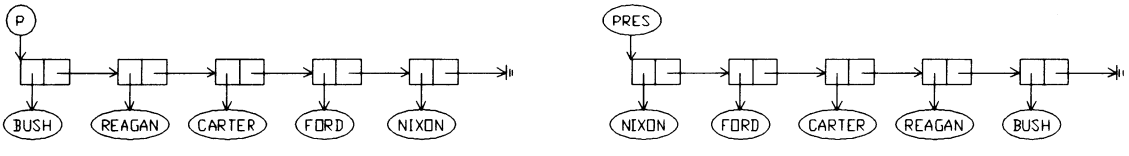


Figure 7-5

```
Command: (third pres)
CARTER
```

The third function returns the third element of a list. We wrote third as a lab exercise in Chapter 5; it isn't a subr. However, once we've written a function it is as much a part of AutoLISP as any of the subrs and can be used in the same manner.

Another good list function that we wrote as a lab in Chapter 6 is nthcdr, which "cdrs" a list the specified number of times. We can add third and nthcdr to our acad.lsp file, and thereafter they will always be present during our drawing sessions.

4. To add clinton to the front of the list we create a cons whose car is clinton and whose cdr is the original list, like so:

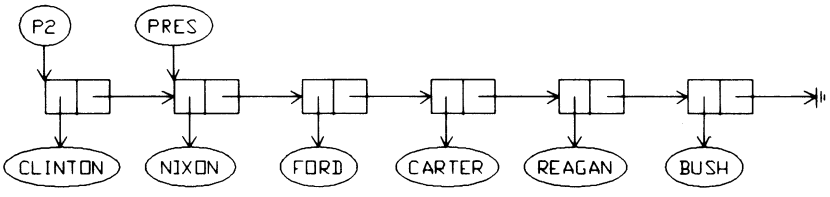


Figure 7-6

What function do we use to create this cons? Right, the `cons` function. We use it as follows:

```
Command: (setq p2 (cons 'clinton pres))
(CLINTON NIXON FORD CARTER REAGAN BUSH)
```

Notes

- We must assign the new cons to a variable, such as `p2`, or it will be lost. We could reuse the symbol `pres`, in which case we'd lose access to the original list. Our choice depends on the requirements of the particular situation.

- We must quote the symbol `clinton` or its value, `nil`, will be used:

  ```
  Command: (setq p2 (cons clinton pres))
  (NIL NIXON FORD CARTER REAGAN BUSH)
  ```

  We cannot quote the symbol `pres` or that symbol becomes the cdr of this new cons:

  ```
  Command: (setq p2 (cons 'clinton 'pres))
  (CLINTON . PRES)
  ```

  We want the symbol's value in this case, rather than the symbol itself. In fact this form has one operand that *must* be quoted and another operand that *cannot* be quoted. Once again, the decision whether or not to quote an operand is never arbitrary.

- We cannot use the list function to add `clinton` to the front of the list:

  ```
  Command: (setq p2 (list 'clinton pres))
  (CLINTON (NIXON FORD CARTER REAGAN BUSH))
  ```

  `list` creates a list comprising its two arguments, the symbol `clinton` and the list `(nixon ford carter reagan bush)`, rather than the five-element list we want.

- Elements can be added to the end of a list, although this is not as easily accomplished. As surprising as it may sound, we usually don't care about the order of elements in a list. Why take the time to cdr to the end of the list just to add an element, when we can simply cons it onto the front of the list? Thus, there is no subr that specifically performs this operation. We'll add an element to the end of a list in Lab Exercise 1.

- Using the `cons` function to add an element to the front of a list (i.e., "consing it onto the list") is a *programming cliché.* We perform this task often enough that we do it by rote rather than thinking about what is actually being done. It is worthwhile adding this programming cliché to your toolbox.

5. Create the list in Figure 7-7:

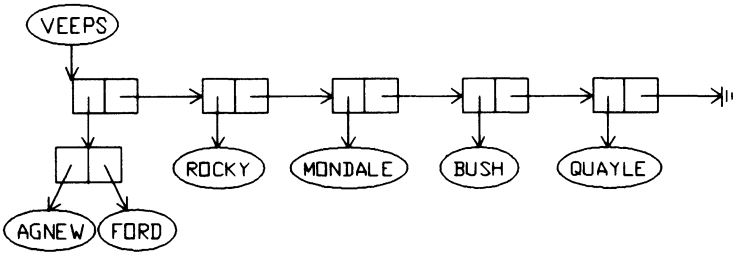


Figure 7-7

```
Command: (setq veeps '((agnew . ford) rocky mondale bush quayle))
((AGNEW . FORD) ROCKY MONDALE BUSH QUAYLE)
```

This example shows a good use for a dotted pair instead of a list. Nixon had two vice presidents, Agnew and Ford. From now until the end of time he will always have exactly two vice presidents.

Whenever we need to store precisely two pieces of related data, a cons is more efficient than a list. However, it is only useful when we know that we will never need to add a third datum. A cons is *not* appropriate for storing 2D points, because we often need to add the Z coordinate. This is why 2D points are stored in lists rather than conses (thus making the Y coordinate be the *cadr* rather than the *cdr*).

Following is the *most important point* in this chapter:

> Neither `agnew` nor `ford` is a member of the list `veeps`.

Remember that the elements of a list are the cars of the backbone conses. Neither agnew nor ford fulfill that requirement (we don't trace through arrows!). The first element of the list veeps is a cons; agnew and ford are the car and cdr of that cons. Functions that operate on members of lists will not access agnew or ford; they will, however, access the cons that points to them.

6. The append function merges the lists pres and veeps into one list, as follows:

```
Command: (setq admin (append pres veeps))
(NIXON FORD CARTER REAGAN BUSH (AGNEW . FORD) ROCKY MONDALE BUSH QUAYLE)
```

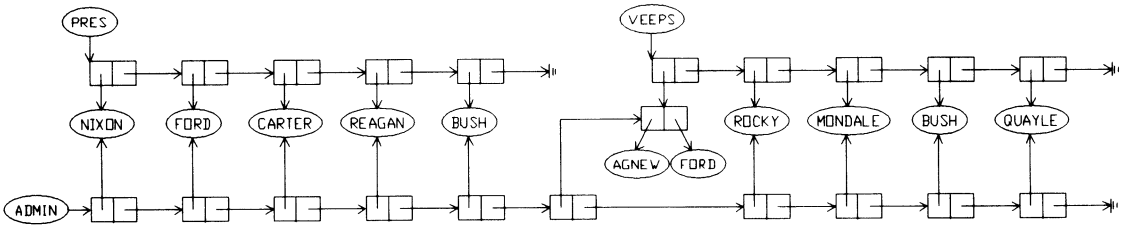


Figure 7-8

append list . . . list	Returns a new list comprising all of the members of the supplied lists in the order specified.

append takes as many arguments as we wish to give to it, but all arguments must be lists. It creates copies of each list and merges them into one long list. Like reverse, append does not alter the original lists; we must setq the new list to a symbol or it will be lost.

Can we use append to add an element to the front of a list?

```
Command: (setq p2 (append 'clinton pres))
error: bad argument type
```

Remember, all arguments must be lists. Given that restriction, we can create a list containing an AutoLISP object and append that to the front of a list in one of the following ways:

```
Command: (setq p2 (append (list 'clinton) pres))
(CLINTON NIXON FORD CARTER REAGAN BUSH)
```

or

```
Command: (setq p2 (append '(clinton) pres))
(CLINTON NIXON FORD CARTER REAGAN BUSH)
```

However, it's simpler and more conventional to use `cons` to add an element to the front of a list.

Occasionally we use `append` to make a *list* be a member of another list. Suppose we have a list of the first three U.S. presidents:

```
Command: (setq first3 '(washington adams jefferson))
(WASHINGTON ADAMS JEFFERSON)
```

We'd like to add this entire list to the end of `pres`. If we enter

```
Command: (setq pres (append pres first3))
(NIXON FOR CARTER REAGAN BUSH WASHINGTON ADAMS JEFFERSON)
```

the presidents are added but *not* as a sublist. To keep the sublist intact we must supply it as a list:

```
Command: (setq pres (append pres (list first3)))
(NIXON FOR CARTER REAGAN BUSH (WASHINGTON ADAMS JEFFERSON))
```

When we begin working with lists of entity data, we'll encounter situations in which this technique must be used.

7. Following are a few ways we can access `rocky`:

```
Command: (car (cdr (cdr (cdr (cdr (cdr (cdr admin)))))))
ROCKY

Command: (cadddr (cdddr admin))
ROCKY

Command: (nth 6 admin)
ROCKY
```

We could also write a `seventh` function that would work analogously to the `third` function we wrote earlier. An even easier way would be to access `rocky` through the symbol `veeps`:

```
Command: (cadr veeps)
ROCKY
```

Recall that the only way to change the value of a variable is with `setq`. Since we haven't altered `veeps`, its value is still the original list of vice presidents (see Figure 7-8). Sometimes it's easier to access structure via old symbols than by symbols we have just created. But we must remember which symbols are still available for our use.

8. We can substitute one list element for another with the `subst` function.

`subst` obj1 obj2 list	Returns a list with all occurrences of obj2 replaced by obj1 in the original list. If obj2 is not a member of the list, `subst` returns a copy of the original list.

`subst` takes three arguments. The first two can be any AutoLISP objects but the third must be a list. It substitutes the first object for all occurrences of the second in the given list. Because `subst` creates and returns a copy of the list rather than altering the original, we must use it with `setq` or we will lose the changes.

You might find it easier to understand this function by reading it from back to front. For example, the following form searches the list `admin` and replaces all occurrences of `ford` with `reagan`. It then rebinds the symbol `admin` to the new list.

```
Command: (setq admin (subst 'reagan 'ford admin))
(NIXON REAGAN CARTER REAGAN BUSH (AGNEW . FORD) ROCKY MONDALE BUSH QUAYLE)
```

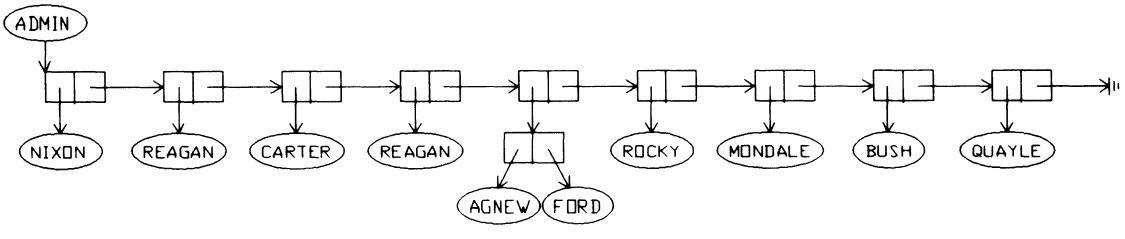


Figure 7-9

As Figure 7-9 shows, only one substitution is performed, because `ford` is a member of the original `admin` list only once. The dotted pair `(agnew . ford)` is a member of `admin`, but its car and cdr, `agnew` and `ford`, are not. Therefore, the `subst` function does not replace the second instance of `ford` with `reagan`.

If we want to replace `ford` in the dotted pair with `reagan`, we must substitute the entire dotted pair, like so:

```
Command: (setq admin (subst '(agnew . reagan) '(agnew . ford) admin))
(NIXON REAGAN CARTER REAGAN BUSH (AGNEW . REAGAN) ROCKY MONDALE BUSH QUAYLE)
```

This code is significant because AutoCAD's entity database is composed of lists of conses. When we want to change something about an entity, such as its layer, we use `subst` to replace the entire dotted pair. In the following section we see how `subst` is used with association lists.

Note that `subst` replaces *all* occurrences of the second argument with the first in the list. Thus if we now enter

```
Command: (setq admin (subst 'rrr 'reagan admin)
(NIXON RRR CARTER RRR BUSH (AGNEW . REAGAN) ROCKY MONDALE BUSH QUAYLE)
```

both occurrences of `reagan` are changed to `rrr`. Again, the occurrence of `reagan` in the cons `(agnew . reagan)` is not affected.

We can locate a specific element of a list with the `member` subr.

`member` obj list	Returns the remainder of the supplied list beginning with the first occurrence of the object, or `nil` if the object is not found.

```
Command: (member 'rocky veeps)
(ROCKY MONDALE BUSH QUAYLE)
```

In this example `member` returns the portion of `veeps` beginning with the element `rocky`. In other words, it returns the cons whose car is `rocky`:

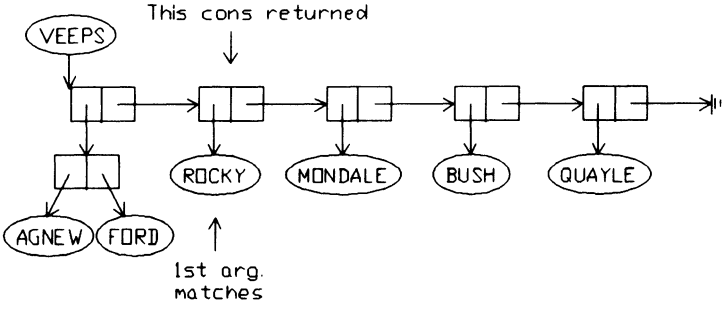


Figure 7-10

When we search for `agnew` in `veeps`, why is `nil` returned?

```
Command: (member 'agnew veeps)
nil
```

This function call returns `nil` because `agnew` is not a member of `veeps`. *The cons whose car is* `agnew` is the element, meaning that the `member` function does not find `agnew` itself. `member` can only access elements, not cars or cdrs of elements. It can, however, locate the entire cons:

```
Command: (member '(agnew . ford) veeps)
((AGNEW . FORD) ROCKY MONDALE BUSH QUAYLE)
```

We can locate the second occurrence of an object in a list by running `member`, finding the cdr of the returned list, and running `member` a second time:

```
Command: (setq sub (member 'c '(a c b c d)))
(C B C D)

Command: (member 'c (cdr sub))
(C D)
```

7.2 Association Lists

An *association list* is a list whose every member is a cons. In general each member is a dotted pair, but occasionally one or more conses are lists.

We typically use association lists to associate or join pairs of items that bear some relationship to each other. For example, association lists are used to store AutoCAD entity data. The car of each pair is a group code and the cdr of each pair is the value for that code. The use of association lists with entities is explained in Chapter 10.

Association lists are typically structured such that the car of each pair is a *key*, and the cdr is the key's *value*. Consider the following association list:

```
Command: (setq part-1 '((name . box) (width . 4) (height . 9) (color . "blue")))
((NAME . BOX) (WIDTH . 4) (HEIGHT . 9) (COLOR . "blue"))
```

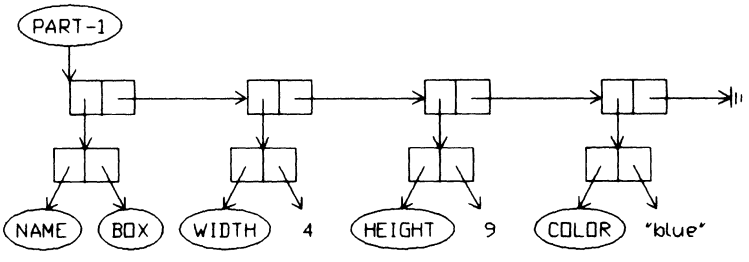


Figure 7-11

The value of `part-1` is a list whose elements are dotted pairs. The key, `name`, has a value, `box`; the key, `width`, has a value, 4, and so forth. Furthermore, there will most likely be another association list that is the value of `part-2`, which also has the name `box` but has a `width` of 3, a `height` of 6.5, and a `color` of `"green"`. In fact, we'd probably have lists for several boxes, each of which has `name`, `width`, `height`, and `color` keys but has different values for those keys.

Whereas the cars tend to remain the same for a given object type, the cdrs are almost always different.

In terms of the entity database, the information for, say, a line is stored in an association list. Whereas each line has a starting point, an ending point, and a layer on which it resides (these are keyed by the car of each pair), the specific point and layer information differs for each line (this is the information stored in the cdr of each pair).

Because we usually associate just two objects, a cons saves considerable space over a list. However, situations do arise where the cdr of one or more conses is a list instead of a dotted pair:

```
((boards . 3) (wires . 116) (colors red green black))
```

In this example the third key is `color` and its value is the list `(red green black)`. When we enter these as a dotted pair

```
(colors . (red green black))
```

they get converted to a simple list. See Section 2.5 to learn how this conversion occurs.

Regardless of whether the cdr is an atom or a list, the association list pairs are handled in exactly the same manner. Although having an element that is not a dotted pair may look unusual, it does not alter how we access and alter these lists. We'll see an example of this shortly.

There is nothing magical about association lists; they're just lists in which we have agreed to make each element a cons. All of the previously described list functions can be used on them. In addition, there is one more function that works specifically with association lists, `assoc`.

`assoc` obj alist	Returns the first association list pair that has the object as its car.

assoc works analogously to the member function. Recall that member searches a list to find a match for its first argument. When it finds a match, it returns the cons for which that element is the car.

assoc works exactly the same way, except that it knows that each member of the list is a cons. Therefore, instead of looking at each element to find a match, it examines the *car* of each element to see whether it matches its first argument. If it does, assoc performs the same action as member: It returns the cons for which that object is the car. If no match is found, assoc returns nil.

For example, we can use assoc to access the width field of part-1:

```
Command: (assoc 'width part-1)
(WIDTH . 4)
```

assoc searches the association list to find a match for the symbol width. Since width matches the car of the second cons, the cons (width . 4) is returned as the result.

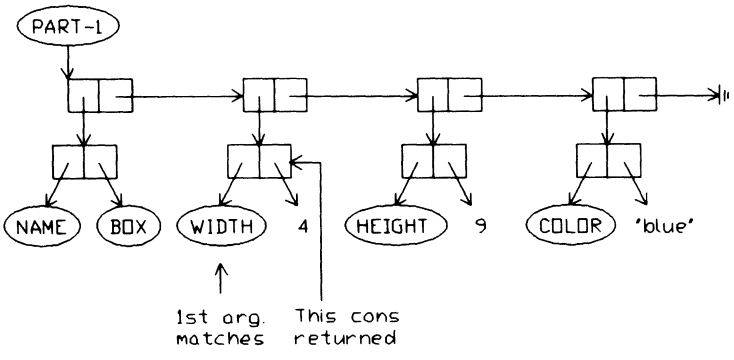


Figure 7-12

Like member, assoc returns the *first* match only. Therefore, we typically create association lists with only one occurrence of each key.

Much of the time we don't really want the pair. Since we know the key (we've supplied that information to assoc), we're really only interested in the value (i.e., the cdr). We obtain the cdr of the pair, of course, with the cdr function:

```
Command: (cdr (assoc 'width part-1))
4
```

Note that if an association list element is a list instead of a cons, we access it in the exact same manner:

```
Command: (setq part-3 '((name . box) (width . 5) (height . 2) (color "cyan" "blue" "red")))
((NAME . BOX) (WIDTH . 5) (HEIGHT . 2) (COLOR "cyan" "blue" "red"))

Command: (assoc 'color part-3)
(COLOR "cyan" "blue" "red")

Command: (cdr (assoc 'color part-3))
("cyan" "blue" "red")
```

We don't need different code to access a pair that's a list; we can safely ignore the difference.

We can replace one pair in an association list with another by using the `subst` function. Remember that `subst` scans the supplied list and replaces all occurrences of its second argument with its first argument. If the second argument is not found, `subst` returns the list unchanged. Since `subst` returns a *copy* of the list, we nearly always use it with `setq`.

Suppose we want to change the `height` field of `part-1` from 9 to 7. We can use `subst` as follows:

```
Command: (setq part-1 (subst '(height . 7) '(height . 9) part-1))
((NAME . BOX) (WIDTH . 4) (HEIGHT . 7) (COLOR . "blue"))
```

This says, "In the list `part-1`, replace the pair `(height . 9)` with the pair `(height . 7)`." When we manipulate entity data, we often use similar code to change, say, the starting point of a line or the wording of some text.

Substituting in this manner can lead to a problem. Suppose that instead of 9 the original height of `part-1` is 9.4756639246. Due to the limits we set in our UNITS command, the `height` might print as 9.47566. When we run the `subst` function using 9.47566, what happens?

```
Command: (setq part-1 (subst '(height . 7) '(height . 9.47566) part-1))
((NAME . BOX) (WIDTH . 4) (HEIGHT . 9.47566) (COLOR . "blue"))
```

In this example 9.47566 doesn't *exactly* match the value for `height`, so the substitution is not made.

Our problem is that we can't explicitly specify `height`'s value because we don't know it. Instead, we must run a function that will return `height`'s value for us. The function that does this is `assoc`:

```
Command: (assoc 'height part-1)
(height . 9.47566)
```

The cdr of this cons is actually the entire value of `height`, but, again, only the first few digits are printed because of UNIT's settings.

In our subst function we use assoc to obtain the height, rather than expressing it explicitly:

```
Command: (setq part-1 (subst '(height . 7)
1>                           (assoc 'height part-1)
1>                            part-1))
((NAME . BOX) (WIDTH . 4) (HEIGHT . 7) (COLOR . "blue"))
```

This says, "In the list part-1, find the pair whose car is height—*I don't care what its cdr is*—and replace it with the pair (height . 7)."

This code is more general than in the previous example, and therefore more powerful. We can use the same code in a loop to change, say, the color of all of our boxes to "yellow". The only information that we don't know about each box is its original color, and by using assoc in conjunction with subst we never need to learn that useless information.

This last example is another programming cliché that we must get to know because we use it quite frequently. As you might imagine, we often want to change real number values, such as the radii of circles. This is the code that accomplishes that task.

7.3 Advanced Functions: apply, mapcar, and lambda

A few functions are somewhat more involved than the ones hitherto shown. If this is your first exposure to AutoLISP, you might wish to come back to these later.

apply func list	Takes a function and a list as arguments and runs the function on the elements of the list.

We can use the apply function to multiply all of the numbers in a list in the following manner:

```
Command: (apply '* '(2 3.5 -4))
-28.0
```

Notice that the function name, *, *must* be quoted.

The symbol must be quoted because apply can also take a lambda expression as its first argument. The lambda must be quoted or it will be executed at the wrong time. The symbol is quoted in this case for consistency.

Subrs such as addition and multiplication permit us to specify as many arguments as we wish, but AutoLISP doesn't provide that capability for our own functions. One way to furnish unlimited data to a function is to have the user supply the data in a list. Given a list, we can use `foreach` to access its elements. In some cases, however, `apply` provides an easy alternative.

Example 1

Suppose we write a function that connects three points into a triangle:

```
(defun lin (p1 p2 p3)
  (command "line" p1 p2 p3 "c"))
```

We might have numerous lists of points such as this one:

```
Command: (setq pts '((2 2) (3 3) (4 1)))
((2 2) (3 3) (4 1))
```

We can put triangles into our drawing by applying the `lin` function to each list of points, as follows:

```
Command: (apply 'lin pts)
nil
```

Example 2

The following function multiplies its second and third arguments, then adds the first:

```
(defun math (x y z)
  (+ x (* y z)))
```

We can call this function in our normal manner:

```
Command: (math -5 4 7)
23
```

However, if we have a list of numbers rather than three separate numbers, then it might be easier to apply the `math` function to this list, as follows:

```
Command: (apply 'math '(-5 4 7))
23
```

`mapcar` func list . . . list	Takes a function and one or more lists as arguments and applies the function first to the car of each list, then to the cadr of each list, and so on. It constructs and returns a list of its results.

`mapcar` is more powerful than `apply` but has a more complex contract. As with `apply`, the function name must be quoted.

We can run or "map" the square root function on a list of four numbers:

```
Command: (mapcar 'sqrt '(4 9 16 36))
(2.0 3.0 4.0 6.0)
```

Since `sqrt` takes one argument, `mapcar` must be given one list. It applies the `sqrt` function to each element of the list and returns a *list* of the results.

When we map a function that takes two arguments, we must supply two lists:

```
Command: (mapcar 'cons '(a b c d) '(1 2 3 4))
((A . 1) (B . 2) (C . 3) (D . 4))
```

In this example `cons` is mapped over each element of both lists in order and a list comprising all of the results is returned. The result of running `mapcar` is always a single list.

What about such functions as addition and multiplication that accept an unlimited number of arguments? `mapcar` takes an arbitrary number of lists for these functions:

```
Command: (mapcar '+ '(3 5 7 9) '(2 6 8 10) '(1 7.0 -3 3))
(6 18.0 12 22)
```

In this example we map + to 3, 2, 1, then 5, 6, 7.0, then 7, 8, –3, then 9, 10, 3. A list of the sums is returned.

If the supplied lists are of different lengths, the function is mapped across the shortest list, and the remaining elements in the longer lists are ignored:

```
Command: (mapcar '+ '(3 5 7 9) '(2 6) '(1 7.0 -3 3))
(6 18.0)
```

Since the second list only has two elements, the third and fourth elements of the other lists aren't used.

Example 1

Earlier we wrote a function named `lin` that connects three points in a triangle:

```
(defun lin (p1 p2 p3)
  (command "line" p1 p2 p3 "c"))
```

It probably requires too much overhead to write this function if we only want to apply it to one set of points. With `mapcar` we can apply the `lin` function to an unlimited number of points. The points must be supplied as lists:

```
Command: (setq l1 '((1 1) (7 3) (4 6) (8 4))
1>               l2 '((5 5) (6 4) (0.5 8) (11 1))
1>               l3 '((5 7) (6 6) (1 2) (10 7)))
((5 7) (6 6) (1 2) (10 7))

Command: (mapcar 'lin l1 l2 l3))
(nil nil nil nil)
```

Remember that `mapcar` operates first on the car of each list. This example creates four triangles: the first connects the points 1,1 to 5,5 to 5,7; the second connects 7,3 to 6,4 to 6,6, and so forth:

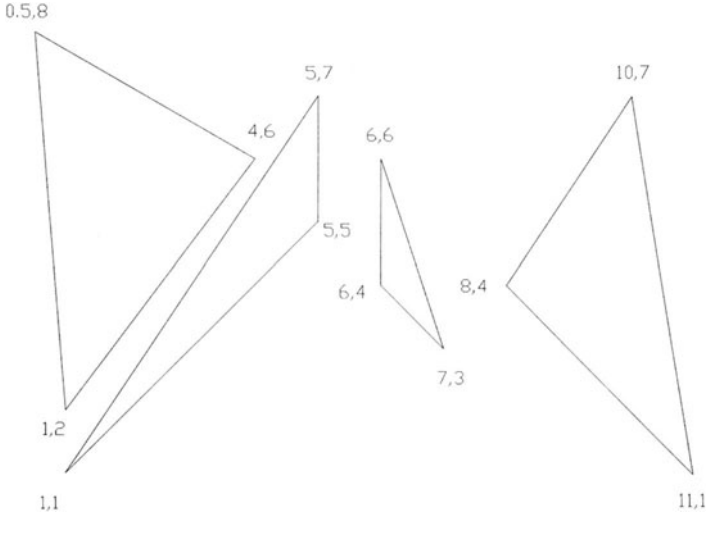


Figure 7-13

Note also that the `command` function always returns `nil`. Since `mapcar` builds a list of the results, this example returns the list (`nil nil nil nil`). In this

case we use mapcar for its effect on the AutoLISP world; we don't care about the result that it returns.

Example 2

Recall the math function that we wrote earlier:

```
(defun math (x y z)
  (+ x (* y z)))
```

We used apply to run this function on a list of numbers. Now let's use mapcar to map our math function on several lists of numbers, as follows:

```
Command: (mapcar 'math '(2 2 2 2) '(1 2 3 4) '(4 5.5 -8 3.0))
(6 13.0 -22 14.0)
```

Because the math function requires three arguments, mapcar must be given three lists. It doesn't matter how many elements are in these lists. Since there are four elements in each list, there will be four numbers in the resultant list.

The car of the second list, 1, is multiplied by the car of the third list, 4, then added to the car of the first list, 2. This gives the result, 6. The same operations are performed on the cadrs, the caddrs, and the cadddrs of these lists, and a list of the results is returned.

lambda list body	Takes a list of parameters and a body and defines a function that performs the tasks specified in the body. It returns the function.

lambda allows us to define a function that we plan to use only once, without giving it a name. This is called an *anonymous* function.

lambda is used in place of defun when we only need to perform the specified tasks once. A defun stays in memory after the function has been run, whereas the space used by lambda is freed up once the function is completed. Furthermore, the function is defined right where we use it, rather than at a different place in our program. This makes the code easier to locate.

There isn't very much we can do with lambda that we cannot do with a normal defun. However, we customarily use it in conjunction with apply or mapcar because it simplifies our use of these functions.

Example 1

Let's look once more at the `math` function:

```
(defun math (x y z)
  (+ x (* y z)))
```

We used this function with `apply` in the following manner:

```
Command: (apply 'math '(-5 4 7))
23
```

If we only use the `math` function in one place, why bother to permanently define it with `defun` and waste space in memory? We can write the code in-line with `lambda`:

```
Command: (apply '(lambda (x y z) (+ x (* y z)))
1>              '(-5 4 7))
23
```

Furthermore, we can still map this function over several lists of numbers by using `mapcar`:

```
Command: (mapcar '(lambda (x y z) (+ x (* y z)))
1>               '(2 2 2 2) '(1 2 3 4) '(4 5.5 -8 3.0))
(6 13.0 -22 14.0)
```

The appearance and functionality of `lambda` is quite similar to that of `defun`; it has a parameter list (to which we can supply additional local variables after a forward slash) and a body. It differs in that we do not give it a name. Like `defun`, `lambda` is a special operator. It's the last one in AutoLISP; every function we see from now on goes through the normal `eval` process.

Since the `lambda` has three parameters (`x`, `y`, and `z`), it must be applied to three arguments. The second argument to the `apply` function, `(-5 4 7)`, and the three lists supplied to `mapcar` satisfy this requirement. As with `defun`, there is a one-to-one matchup between parameters and arguments.

Example 2

Here's a situation where `lambda` comes in quite handy. A lab in Chapter 5 had us supply two points and calculate a midpoint. We can do this more efficiently by using `mapcar` and `lambda`:

```
(defun midpoint (pt1 pt2)
  (mapcar '(lambda (a b) (/ (+ a b) 2.0)) pt1 pt2))
```

We would run this function as follows:

```
Command: (midpoint '(2 4 0) '(6 9 2))
(4.0 6.5 1.0)
```

`midpoint`'s parameters, `pt1` and `pt2`, are bound to the arguments, `(2 4 0)` and `(6 9 2)`. Since `mapcar` operates first on the car of each list, `lambda`'s parameters, `a` and `b`, are initially assigned 2 and 6, respectively. Their values are added and the sum is divided by 2, producing the first result, 4.0.

This process is repeated for 4 and 9, then 0 and 2. `mapcar` finally constructs and returns a list of the results. (`midpoint` was written by Todd Moen.)

7.4 Important Points to Remember

- Use `car`, `cdr`, and `cadr` to access the X, Y, and Z coordinates of a point. Never use the `last` function to access any of these. Who knows when AutoCAD will go to 4D!

- Members of a list are the cars of the backbone conses. An atom in a sublist is *not* an element of the parent list. Given the list `(1 2 3 (4) 5)`, list manipulation functions will not access the number 4.

- If we intend a list to be data, we must quote it so that it's not evaluated. If the list is code, then we don't quote it so that it is evaluated.

- We typically create a list in one of the following two ways:

```
Command: (setq list1 '(a b c))
(A B C)

Command: (setq list2 (list 'a 'b 'c))
(A B C)
```

  We use the former method when we can explicitly supply the data that will become the list's members. We use the `list` function when one or more operands need to be evaluated. We build conses in an analogous manner.

- Functions that appear to alter existing list or cons structure never do; they always create copies that reflect the changes. Use these functions inside of a `setq` to save the new version.

- Here are several programming clichés. Become familiar with these because we use them all the time:

  - Cons (add) the symbol x onto the front of the list `l`:

    ```
    (setq l (cons 'x l))
    ```

    When we build a list in this manner, the last element entered is at the front of the list. When the list is completed, enter `(setq l (reverse l))` to put the elements in the correct order.

  - Learn the value of the 8 field in the association list, `al`:

    ```
    (cdr (assoc 8 al))
    ```

  - Substitute the pair `(8 . "0")` for the current 8 field in the association list, `al`:

    ```
    (setq al (subst '(8 . "0")
                    (assoc 8 al)
                    al))
    ```

- The first argument to `apply` and `mapcar` is either a function name or a `lambda` expression. In either case it must be quoted.

7.5 Labs

Many of these exercises can be solved using the iteration functions we learned in the last chapter. A problem statement is followed by (rec) if the Answers appendix shows a recursive solution as well. It is followed by (lambda) if it is also solved using a `lambda` function.

1. Write the code to accomplish each of the following tasks. Each step depends on results from the previous step, and the expected result is listed below the problem.

   Notes

   - It's easier to enter this exercise at the Command: prompt rather than creating functions in the editor, but be sure to write down the code used to obtain each solution.

   - We also suggest binding the result of step a) to the variable `sa`, the result of step b) to `sb`, and so forth. This makes it easier to retrace your steps if you make a mistake.

- Some examples might take several steps. There's often more than one way of accomplishing the same task.

a) Create a list containing, in order, abbreviations for the states Massachusetts, New York, Georgia, Washington, and California.

```
(MA NY GA WA CA)
```

b) Remove Massachusetts from the list.

```
(NY GA WA CA)
```

c) Remove California from the list (this one's tricky!).

```
(NY GA WA)
```

d) Add Ohio to the front of the list.

```
(OH NY GA WA)
```

e) Add Ohio to the end of the list.

```
(OH NY GA WA OH)
```

f) Can you think of a way to copy the last element of this list (whatever it may be) to the front of the list?

```
(OH OH NY GA WA OH)
```

g) Create another list containing British Columbia, Ontario, and Quebec.

```
(BC ONT QUE)
```

h) Merge the two lists into a single new list.

```
(OH OH NY GA WA OH BC ONT QUE)
```

i) Copy the seventh element of the new list to the front of this list.

```
(BC OH OH NY GA WA OH BC ONT QUE)
```

j) Replace all occurrences of Ohio with Texas.

```
(BC TX TX NY GA WA TX BC ONT QUE)
```

k) Return the portion of the list beginning with Georgia.

```
(GA WA TX BC ONT QUE)
```

2. Write a function that takes a list and an object as arguments. Have it add the object to the end of the list.

```
Command: (add-end '(post1 post2 post3) 'post4)
(POST1 POST2 POST3 POST4)
```

3. Write a function that adds a pair to the front of an association list.

```
Command: (setq al '((a . 1) (b . 2) (c . 3)))
((A. 1) (B . 2) (C . 3))

Command: (setq al (add-pair '(d . 4) al))
((D . 4) (A . 1) (B . 2) (C . 3))
```

4. Enter the following form at the Command: prompt. Remember that, depending on our UNITS settings, an entire number may not be displayed.

```
Command: (setq al '((a . "xyz") (b . 1.23456789) (c 7.654321 9.876789)))
((A. "xyz") (B . 1.23457) (C 7.65432 9.87679))
```

At the Command: prompt, enter the code to change the first pair in this association list to (a . "jkl"):

```
((A . "jkl") (B . 1.23457) (C 7.65432 9.87679))
```

Next, enter the code to change the second pair in this new list to (b . 1.3):

```
((A . "jkl") (B . 1.3) (C 7.65432 9.87679))
```

Finally, alter the last pair so that the association list appears as follows:

```
((A. "jkl") (B . 1.3) (C 7.0 9.0))
```

5. Write a function that takes an AutoLISP object and a list and returns a new list with all occurrences of the supplied object removed.

```
Command: (remove 2 '(1 2 a 2 12 "jkl" 2 7))
(1 A 12 "jkl" 7)
```

6. Write a function that takes an AutoLISP object and a list and returns a count of the number of times that object is a member of the list (rec).

```
Command: (count 3 '(2 a 3 b 3 c 4))
2
```

7. Write a function that takes a number and a list of numbers and multiplies each element of the list by the first argument. Return a list of the results (lambda).

```
Command: (multby 3 '(2 4.3 -6))
(6 12.9 -18)
```

8. Write a function that takes a list and returns a copy of that list (rec).

```
Command: (setq j '(x y z))
(X Y Z)

Command: (setq k (cop j))              ;copy the list
(X Y Z)

Command: (eq j k)
nil
```

9. Write a function that takes a list of integers and returns a list of the even ones (lambda).

```
Command: (evens '(1 5 2 4 7 6 -22 3))
(2 4 6 -22)
```

10. Write a function that takes a list and builds a new list consisting of every other member of the original list (rec).

```
Command: (other '(2 a 3 b 3 c 4))
(2 3 3 4)
```

11. Write a function called `assoc2` that works exactly like `assoc` except that it keys on the cdr of the sublist rather than its car. Have it return the first occurrence of the key.

```
Command: (setq l '((a . 1) (b . 2) (c. 1)))
((A . 1) (B . 2) (C . 1)))

Command: (assoc2 1 l)
(A . 1)
```

12. Write a function that takes two lists of arbitrary length and creates an association list in the following manner:

```
Command: (map-al '(a b c) '(1 2 3 4))
((A . 1) (B . 2) (C . 3))
```

Hint: Use `mapcar` and `cons`.

13. In Chapter 6, Lab Exercise 13 we wrote a function to find the sum of a list of numbers. Given the functions introduced in this chapter, can you think of one or more additional ways to accomplish this task?

14. In Chapter 6, Lab Exercise 19 we wrote a function called `put-strs`. Rewrite this lab using `lambda`.

15. Write a function that takes a list of points and a string indicating whether to draw a LINE or a PLINE. Draw an open LINE or a closed PLINE connecting the specified points. This is a challenging exercise that may require the use of `apply` or an explicit call to `eval`.

```
Command: (connect '((2 2) '(3 4) '(1 2.4)) "pline")
```

    The function draws a closed polyline connecting these points.

16. Modify the preceding function so that lines are drawn on Layer 0 and polylines are drawn on Layer 1.

8

Input/Output

Topics Covered in This Chapter

- `get` functions and `initget`
- Read and write functions
- I/O to files
- The `s::startup` function
- Replacing standard AutoCAD commands with our own commands
- Obtaining filenames via dialog boxes
- Advanced file access

Goals for This Chapter

- Learn how to use the various input and output functions to get user input and to write out results to the screen.
- Learn AutoLISP's elegant user interface, which enables our commands to look like built-in AutoCAD commands. See how to use `initget` to limit or extend the types of input that the user can provide.
- Understand the concepts of "opening" and "closing" files. Learn how to write our data to files so that they can be read back in during another drawing session.
- Discover more advanced ways of accessing files.
- Learn advanced techniques for loading code and variables into our AutoLISP environment.

Introduction

Of all the features Autodesk added to the AutoLISP language, the `get` functions might be the sweetest. They enable us to tailor our AutoCAD commands to look exactly like the built-in AutoCAD commands. Furthermore, they're extremely intuitive and easy to use.

AutoLISP also has standard I/O functions that allow us to read and write to files.

With nearly every release of AutoCAD, AutoLISP's I/O capabilities increase. In this chapter we also explore some of AutoLISP's advanced I/O features.

Note: You will more readily understand the functions in this chapter if you type in the examples as you encounter them.

8.1 The `get` Functions: Building an Elegant User Interface

The `get` functions all work similarly in that they request a specific kind of input, then halt processing until the user enters the proper data. If the user enters the wrong kind of data, they repeatedly reprompt until the correct input is entered.

The `get` functions all take an optional prompt. A *prompt* is a string that is displayed on the user's screen when the function is executed. We use it to specify the datatype that the user should enter.

Although the functions `getvar`, `getenv`, and `getfiled` have names that begin with *get,* they are not considered `get` functions.

`getint` [str]	Returns an integer entered by the user.

In its simplest form, we can run the `getint` function by entering

```
Command: (getint)
```

When the user enters an integer, `getint` returns that integer, like so:

```
Command: (getint) 5
5
```

Remember that when we enter code at the Command: prompt, our results are returned to the screen. This means that in many of this chapter's examples we will see our data twice. When we run these functions within a `defun`, we will only see the data once.

When we run a `get` function, we usually provide a prompt string so that the user knows what kind of data to enter. Furthermore, since we generally want to save what the user types, we typically put a `get` function inside of a `setq`:

```
Command: (setq num (getint "Enter an integer between 0 and 100: "))
Enter an integer between 0 and 100: 23.89

Requires an integer value.
Try again: 56
56
```

`getint` demands that we enter an integer value and reprompts until we do so.

Notice that the colon and space at the end of our prompt make it easier to read. These won't be in the prompt unless we put them there. Whatever text we place between the double quotes in the prompt is printed when we run the function:

```
Command: (getint "Enter a number")
Enter a number12
12
```

`getint` is typical of the `get` functions in that it limits the user's input to a specific datatype. If, however, instead of entering the required datatype the user presses `<enter>`, the `get` functions (except `getstring`) return `nil`:

```
Command: (getint "Enter an integer: ")
Enter an integer: (User hits <enter>)
nil
```

This means that the `get` functions are good candidates for tests to a `while` loop or an `if`. We can continually run a loop until the user presses `<enter>`, whereupon the `get` function returns `nil` and the loop exits:

```
(while (getint "Enter an integer: ")
    .
    .
    .)
```

See the `c:long-dist` command in Section 6.3 for a program example showing this technique.

We cannot supply an AutoLISP expression as input to a `get` function:

```
Command: (getint "Enter an integer: ")
Enter an integer: (* 2 3)
Can't reenter AutoLISP.
Requires an integer value.
Try again:
```

`getreal` [str]	Returns a number entered by the user.

`getreal` works identically to `getint`, except that it does not restrict the type of number that the user enters. If the user enters an integer, `getreal` converts it to a real number.

```
Command: (getreal "Enter a number: ") 31.456
31.456

Command: (getreal "Enter a number: ") 6
6.0
```

We don't use the `getreal` function very often, because the `getdist` function does everything `getreal` can do, plus more.

`getdist` [pt] [str]	Returns a distance entered by the user.

`getdist` is a powerful function that enables the user to enter a distance in a couple of different ways:

- The user can enter a distance at the keyboard. When used this way, `getdist` is identical to `getreal`:

```
Command: (getdist "Enter a distance: ")
Enter a distance: 8.02
8.02
```

- The user can select two points on the screen and `getdist` returns the distance between them.

`getdist` accepts a point as an optional first argument. If one is supplied, `getdist` draws a rubber-band line from that point to the current crosshairs position and waits for the user to enter a second point by typing or clicking. It then

calculates and returns the distance between the two points. When we use getdist in this fashion, we typically prompt for the second point:

```
Command: (getdist '(3 4) "Enter second point: ")
Enter second point: 3,6.5
2.5
```

We can specify a distance by supplying a number in the current distance units format, such as feet and inches. However, the distance is always returned as a real number:

```
Command: (getdist "Enter distance: ")
Enter distance: 3'7-1/2"
43.5
```

getdist returns a 3D distance unless initget bit 64 is set.

We can use snaps when responding to any get function that requests a point. Let's draw a line from 3,3 to 5,3:

```
Command: (command "line" '(3 3) '(5 3) "")
nil
```

Now let's ask the user to input a distance:

```
Command: (getdist '(4 7) "Enter 2nd point: ")
Enter 2nd point: mid of (select line) 4.0
```

Our ability to snap to points makes getdist the function of choice in many situations where we might otherwise use getreal. Suppose we want the user to enter the length of a piece of pipe. We could use getreal for this, but getdist is more powerful. By using snaps the user can effectively say, "I want this section of pipe to have the same length as that section," without having to know the pipe's actual length.

getpoint [pt] [str]	Returns a point entered by the user.

getpoint works quite similarly to getdist. It displays the optional prompt, waits for the user to type in or click on a point, then returns that point.

As with getdist, getpoint takes an optional base point argument. If one is supplied, getpoint draws a rubber-band line from that point to the current crosshairs position. Whereas getdist returns the distance between the base

point and the selected point, `getpoint` returns the selected point only; the base point is used solely as a visual cue.

```
Command: (setq p (getpoint '(2.2 3.0) "Second point: "))
Second point: 4,6
(4.0 6.0 0.0)
```

We use `getpoint`'s base point in the `long-dist` command (shown in Section 6.3) to pinpoint where we are so we can decide where to click next. But the base point is ultimately irrelevant to the point that is returned by `getpoint`.

`getcorner` pt [str]	Displays a box prompt and returns a point entered by the user.

`getcorner` works almost identically to `getpoint`. It displays the optional prompt, waits for the user to type in or click on a point, then returns that point. There are two differences between `getcorner` and `getpoint`:

- The base point is required by `getcorner`.
- `getcorner`'s prompt is a box instead of a rubber-band line.

We use `getcorner` when we intend to draw a box. By displaying a box rather than a rubber-band line, `getcorner` makes it easier for the user to determine where to click. Like `getpoint`, `getcorner` returns the selected point; the box is a visual guide only.

```
Command: (getcorner '(4 5) "Enter opposite corner: "))
Enter opposite corner: 7,3
(7.0 3.0 0.0)
```

`getstring` [t] [str]	Returns a string entered by the user.

`getstring` is one of the most commonly used `get` functions because it enables our programs to elicit text information from a user. It is used as follows:

```
Command: (setq string (getstring "What's this part called? "))
What's this part called? door
"door"
```

If the string is longer than 132 characters, only the first 132 are returned.

In order for the get functions to allow us to tailor our own AutoCAD commands to look like the built-in ones, they themselves must work like standard AutoCAD commands. When we supply data to AutoCAD, we can terminate our input by hitting the <space> or <enter> keys. The get functions must have a similar interface, which leads to a little problem.

If <space> is a terminating character to getstring, it means that the strings the user enters cannot contain spaces. Suppose the getstring asks the following question and the user wishes to answer "6-foot sliding door." Watch what happens when she does:

```
Command: (setq string (getstring "What's its real name? "))
What's its real name? 6-foot
"6-foot"
```

As soon as the user hits the <space> key, getstring picks up and returns the one-word string that was typed.

We frequently need to limit the user to typing one-word strings, but we just as often want to allow multiword input. To get around this restriction, getstring accepts an optional first argument. If the argument is t, it tells getstring to disregard <space> as a terminating character. Then the user must press <enter> to conclude input:

```
Command: (setq string (getstring t "What's its real name? "))
What's its real name? 6-foot sliding door
"6-foot sliding door"
```

We earlier noted that if the user presses <enter> in response to a get function, the function returns nil. This attribute makes the get functions good tests for the while loop. getstring is the one exception to this rule. When the user presses <enter>, getstring returns the null string instead:

```
Command: (getstring "Type component names. Hit <enter> when done: ")
Type component names. Hit <enter> when done: (User types <enter>)
""
```

We can still use getstring in the while loop's test, but we must specifically check for the null string:

```
(while (not (equal (getstring "Type component names: ") ""))
. . .)
```

When comparing strings, keep in mind that "abc" and "ABC" are not the same.

getstring doesn't honor any of the initget bits.

`getangle [pt] [str]`	Returns an angle entered by the user. The angle is returned in radians.

`getangle` runs in a manner similar to `getdist`. It displays an optional prompt and waits for the user to either type in an angle in the current angle units format (e.g., degrees) or click on two points to measure the angle. If the user either clicks on or types in a single point, `getangle` automatically prompts for the second point:

```
Command: (getangle "Enter angle: ")
Enter angle: 2,2
Second point: 2,4
1.5708
```

As with `getdist`, `getangle` takes an optional base point argument. If one is supplied, `getangle` draws a rubber-band line from that point to the current crosshairs position. It then returns the angle between the base point and the point selected by the user.

The following example draws a rubber-band line from 1,3 to the current crosshairs position, then returns the angle between that point and the point selected by the user:

```
Command: (setq ang (getangle '(1.0 3.0) "Select second point: "))
Select second point: 0,3
3.14159                                                ;180 degrees
```

The angle is always returned in radians with respect to the current construction plane. Remember that we can use our `rtd` function to convert the returned angle to degrees:

```
Command: (rtd (getangle '(1.0 3.0) "Select second point: "))
Select second point: 0,3
180.0
```

`getangle` measures the angle according to the current ANGBASE setting but always measures it in a counterclockwise direction regardless of the current ANGDIR setting. The algorithm that AutoLISP uses to measure this angle is explained in more detail with `getorient`.

`getorient` [pt] [str]	Returns an angle entered by the user. The angle is returned in radians.

`getorient` is run exactly like `getangle` but is used when we need an absolute angle (orientation) instead of a relative angle (rotation amount). If you never change your ANGBASE from due east, you'll never need to use `getorient`. Furthermore, the differences between these two functions are subtle and initially tricky, so most AutoLISP users just get a sense of how `getorient` works, then study its details when they need to use it.

The results returned by `getangle` and `getorient` differ depending on the settings of the ANGBASE and ANGDIR system variables.

ANGBASE is the variable that indicates the base point for angle measurements. The default is 0 (east) and we can set it to, say, 90 (north) with a `setvar` function call, or by running the UNITS command.

ANGDIR indicates the direction in which angles are input. The default is 0 (counterclockwise), and we can set it to 1 (clockwise) with `setvar` or by running the UNITS command.

When ANGBASE is set to 0, `getorient` and `getangle` work identically, and the result that either returns is determined by the setting of ANGDIR.

- When ANGDIR is 0, then the angle is input and measured in our customary way, from due east in the counterclockwise direction:

```
Command: (rtd (getorient "Enter angle: "))
Enter angle: 40
40.0
```

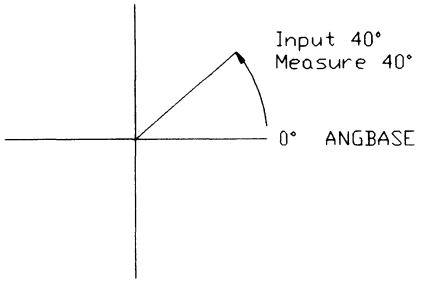


Figure 8-1

- When ANGDIR is 1, the angle is *input* in a clockwise direction but *measured* counterclockwise:

```
Command: (rtd (getorient "Enter angle: "))
Enter angle: 40
320.0
```

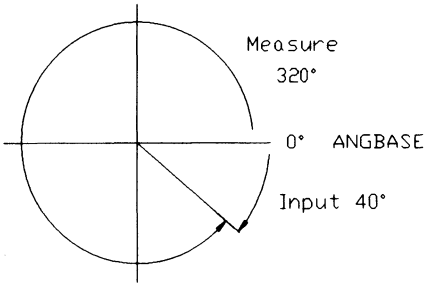


Figure 8-2

Since our ANGBASE setting is east and our input direction is clockwise, the input point is roughly southeast. `getorient` (and `getangle`) returns a measurement from east in a counterclockwise direction, which is 320 degrees.

Reset ANGDIR to 0 (counterclockwise) and set ANGBASE to 90 (i.e., measure from north). Now `getangle` and `getorient` behave quite differently:

- `getangle` inputs a point that is 40 degrees from due north in a counterclockwise direction. It measures that angle, again from the north in a counterclockwise direction. In other words, it returns the same result as it did when ANGBASE was east.

```
Command: (rtd (getangle "Enter angle: "))
Enter angle: 40
40.0
```

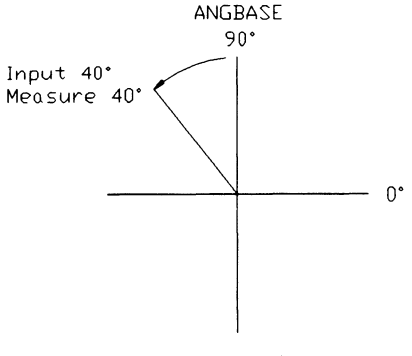


Figure 8-3

- `getorient` also inputs a point that is 40 degrees from due north in a counterclockwise direction. However, it measures from east regardless of the setting of ANGBASE. Given the same circumstances, `getorient` measures 90 degrees from east to north plus the additional 40 degrees.

```
Command: (rtd (getorient "Enter angle: "))
Enter angle: 40
130.0
```

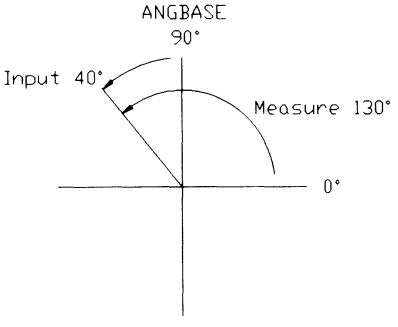


Figure 8-4

Finally, leave ANGBASE at north and reset ANGDIR to 1 (clockwise).

- The user inputs an angle that is 40 degrees from due north in a clockwise direction, that is, northeast. `getangle` measures from north in a counterclockwise direction and returns 320 degrees as the result.

```
Command: (rtd (getangle "Enter angle: "))
Enter angle: 40
320.0
```

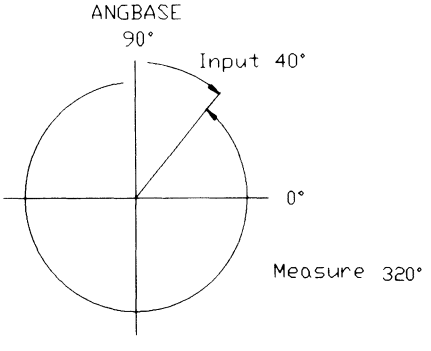


Figure 8-5

♦ Using `getorient`, the user again enters an angle that indicates northeast, which is 50 degrees when measured counterclockwise from east.

```
Command: (rtd (getorient "Enter angle: "))
Enter angle: 40
50.0
```

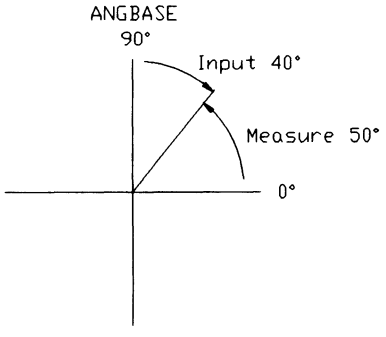


Figure 8-6

Here's one last observation. If the user inputs points rather than an actual angle, both of these `get` functions essentially ignore the ANGDIR setting. Keep in mind that ANGBASE is still north.

```
Command: (rtd (getangle '(1 1) "Enter second point: "))
Enter second point: 2,2
315.0
```

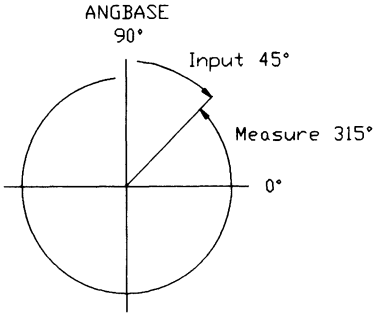


Figure 8-7

```
Command: (rtd (getorient '(1 1) "Enter second point: "))
Enter second point: 2,2
45.0
```

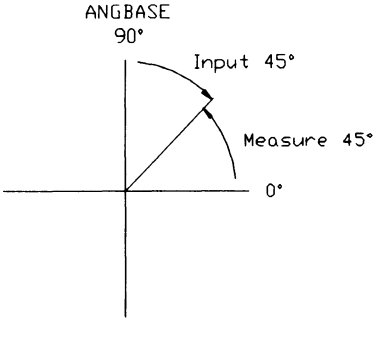


Figure 8-8

The angle from 1,1 to 2,2 measured from due east is 45 degrees; this is what `getorient` returns. `getangle` returns 315 degrees because it inputs and measures from due north as in the previous example. These examples yield the same results regardless of the setting of ANGDIR.

`getkword` [str]	Returns a keyword entered by the user.

We use `getkword` when we want to supply the user with a list of keywords to choose from. The user is then required to select one of the specified keywords (or press the `<enter>` key). User input is not case sensitive, and `getkword` will reprompt if the user enters an unacceptable response. We typically use the optional prompt to list the keywords that the user may enter.

Acceptable keywords must be established by a previous call to the `initget` function. Thus, prior to seeing examples using `getkword`, we must study `initget`.

8.1.1 Using `initget` to Fine-Tune the `get` Functions

`initget` [int] [str]	Accepts bit and keyword arguments that control the user's input to the subsequent `get` function. It returns `nil`.

The name `initget` means ***INIT***ialize the ***GET*** functions. To perform this initialization, `initget` has two powerful and completely independent facilities.

The first optional argument to `initget` is an integer that specifies bit settings. These bits give us greater flexibility by limiting or expanding the type of input that the user can supply to a particular `get` function. The following bit values are accepted by `initget`:

1	Disallow null input
2	Disallow zero values
4	Disallow negative values
8	Don't check limits
16	Not used
32	Use dashed lines for rubber-band or box
64	Disallow input of a Z coordinate for `getdist`
128	Accept arbitrary string input

Here are the meanings of some of these bits:

1	Don't allow the user to press the <enter> key without providing input.
8	Don't limit points being input, even if limit checking is on (i.e., LIMCHECK is set to 1).
32	If numerous entities are already highlighted, it might be hard to see a rubber-band line or a box. This bit makes the prompt be a dashed line so it stands out.
64	Forces `getdist` to perform a 2D measurement.
128	Explained in detail below.

Note: The *AutoLISP Reference* uses two slightly different concepts, *bit* and *bit value.* The preceding chart specifies a *bit value,* which is the number we must enter to select the appropriate option.

Bit values can be added together to effect more than one constraint. For example,

```
Command: (initget 7)
nil
```

sets bit values 1, 2, and 4. We can indicate this more explicitly by showing the actual addition:

```
Command: (initget (+ 1 2 4))
nil
```

We'd use this latter method when setting, say, bit values 32 and 64, and just specifying their sum wouldn't clearly indicate which bit values are set.

A bit value of 7 prevents the following `get` function call from accepting null, 0, or negative input. When we issue the next `getint` call, the user is constrained to enter a positive integer:

```
Command: (getint "How many panes in these windows? ")
How many panes in these windows? (User hits <enter>)

Requires a positive nonzero integer.
How many panes in these windows? 0

Value must be positive and nonzero.
How many panes in these windows? -5

Value must be positive and nonzero.
How many panes in these windows? 3.4

Requires a positive nonzero integer.
How many panes in these windows? 6
6
```

getint continues to prompt until the user has satisfied its restricted input criteria. This gives us tremendous control over the data that the user can input. It's essential too, because we don't want a user to specify that a window has –5 panes!

An initget setting applies only to the next get function call. If we want the bits set a second time, we must reissue the initget. If we rerun the preceding getint function without resetting initget, it *will* accept a negative number:

```
Command: (getint "How many panes in these windows? ")
How many panes in these windows? -5
-5
```

Notes

- Although initgets are frequently used for controlling input, our programs don't usually become clogged with them. The times when we repeatedly issue an initget we typically find ourselves doing so in a loop.

- An initget can be placed anywhere in our program; the bits are set until the next get function is run. We typically place an initget immediately before the get function it modifies, however, to make its purpose apparent.

Not all get functions handle all of the initget bits, as shown in the following chart. For example, it makes no sense for getpoint to disallow negative numbers (its input is a point) or for getint to perform limit checking.

Bit value	*Functions that honor this bit*							
1—No null	**getint**	**getreal**	**getdist**	**getpoint**	**getcorner**	**getangle**	**getorient**	**getkword**
2—No zero	**getint**	**getreal**	**getdist**			**getangle**	**getorient**	
4—No negative	**getint**	**getreal**	**getdist**					
8—No limits				**getpoint**	**getcorner**			
32—Dashed lines			**getdist**	**getpoint**	**getcorner**	**getangle**	**getorient**	
64—2D distance			**getdist**					
128—Arbitrary input	**getint**	**getreal**	**getdist**	**getpoint**	**getcorner**	**getangle**	**getorient**	**getkword**

getstring honors *none* of the initget bits.
All of the get functions honor initget keywords, as do entsel, nentsel, and nentselp.

♦ ♦ ♦

initget has another powerful feature that is separate from and independent of the control bits. It can be used with the get functions (except getstring) to allow the user to enter a keyword or an arbitrary string instead of the expected datatype.

The following example shows how we would use initget with getkword to specify keywords that the user may enter:

```
Command: (initget "BASement BAThroom Bedroom")
nil                                          ;Specify keywords

Command: (getkword "Enter room to modify: ")
Enter room to modify: kitchen                ;User makes invalid choice

Invalid option keyword
Enter room to modify: ba

Ambiguous response, please clarify . . .
BASement or BAThroom? bas                    ;User makes acceptable choice
"BASement"
```

initget takes an optional string argument that specifies keywords that the user may enter. getkword waits for the user to enter one of these keywords, then returns it as a result. getkword reprompts if the user issues an incorrect or ambiguous keyword.

Notes

- It's good programming style to specify the user's choices in the getkword prompt, like so:

```
Command: (getkword "Enter room to modify (<BASement> BAThroom BEdroom): ")
Enter room to modify (<BASement> BAThroom BEdroom):
```

  Now the user doesn't have to guess what input is expected. Note, however, that this prompt in no way affects the choices that the user is allowed to make; they are determined by the initget. The prompt merely helps the user make a choice.

  To conform to AutoCAD's style, we indicate the default selection by placing it inside angle brackets. If the user hits <enter>, we'll assume that "BASement" was the intended response. Once we've gotten the user's input, we must determine which keyword was typed. This is a situation where cond comes in handy.

- When we list the choice string in the `initget`, we put in uppercase enough characters to uniquely identify the choice. This allows the user to type only those characters when making a selection. If the entire string is either upper or lowercase, then the user must type the whole keyword to make a choice.

  We aren't required to make the initial characters uppercase. The following example shows a valid alternative:

```
Command: (initget "baSement baThroom beDroom eXit")
nil
```

  Now the user can select `"beDroom"` simply by typing `"d"`. Although user input can be either uppercase or lowercase, we must specifically check for `"beDroom"`; the strings `"bedroom"` or `"BEDROOM"` won't match.

The two `initget` facilities, bit values and keywords, can be used together. In the following example we again set `initget` bit values 1, 2, and 4, to block null, zero, and negative input. However, the user now has the option of entering `"s"` to indicate that the window is made of stained glass:

```
Command: (initget 7 "Stained-glass")
nil

Command: (getint "How many panes in these windows (Stained-glass)? ")
How many panes in these windows (Stained-glass)? 0

Value must be positive and nonzero
How many panes in these windows (Stained-glass)? s
"Stained-glass"
```

Our program must now determine whether the user entered an integer or a string.

Advanced Note: `initget` can accept a string containing *localized keywords.* These are keywords that are presented to the user but will be translated into *language independent keywords* by `initget`. Observe:

```
Command: (initget "Part1 pArt2 _X1 X2")
nil

Command: (getint "Enter part number (Part1 pArt2): ")
Enter part number (Part1 pArt2): a
"X2"
```

In the `initget` string we place all of our localized keywords followed by the *exact same number* of language independent keywords, the first of which is

preceded by an underscore. When the user selects a keyword from the former category, AutoLISP returns the corresponding keyword from the latter category. The advantage is that different programs, languages, or facilities can present different keywords to the user but have standardized results returned to the program.

Localized keywords were added to AutoLISP in Release 13 C2.

Arbitrary String Input

We set `initget` bit value 128 when we want a `get` function to accept arbitrary string input from the user. It is similar to using `getkword` except that there are no restrestrictions on the string that the user can enter. It differs from `getkword` in that the user cannot enter abbreviations unless we set up our code to explicitly test for them.

In a simple example, bit value 128 works quite similarly to `initget`'s keyword facility:

```
Command: (initget 128)
nil

Command: (getreal "Enter a number: ")
Enter a number: pi
"pi"
```

This is a classic example because there is no way that the user can enter the numeric value for `pi`. Our program can test whether the user entered the string `"pi"`, in which case we'd use the variable `pi`.

`initget` honors the other control bits and keywords before bit value 128. For example, if we also disallow zero input, `getint` won't accept 0, although it will accept 3.6 and return it as the string `"3.6"`. However, if the `initget` bit value of 1 is also set and the user hits `<enter>`, a null string is returned instead of `nil`.

Example

The `getpnt` function shows a practical application using arbitrary string input. It utilizes the `read` function and requires an explicit call to `eval`.

Suppose the user has bound the variables `p1`, `p2`, and `p3` to points that he's previously selected:

```
Command: (setq p1 '(2 3) p2 '(4 5) p3 '(6 7))
(6 7)
```

When our program asks the user to select a point, he'd like to simply type p2 instead of reselecting the point. Setting the initget bit value of 128 allows our program to accept that input:

```
Command: (getpnt)
Select a point: p2
(4 5)
```

The getpnt function accepts either string or point input and returns the point entered by the user:

```
  (defun getpnt (/ pp)
1   (initget 128)
    (setq pp (getpoint "Select a point: "))
2   (if (eq (type pp) 'str)                    ;Did the user enter a string?
        (eval (read pp))                       ;Yes, convert to symbol & evaluate it
3       pp))                                   ;No, just return the selected point
```

1. getpnt first sets initget bit value 128. It then issues the getpoint subr and binds local variable pp to the value entered by the user.

2. Next, it tests whether the user's input is a string. If so, the read function converts the string to a symbol and eval finds the value of that symbol. This value is returned by getpnt.

3. If the input is not a string, getpnt simply returns the value of pp, which is the point that the user entered.

8.2 Other I/O Functions

The get functions are powerful and easy to use, but they have a couple of limitations. First, they're all input functions. Second, they permit console input only; they can't read from files.

The other, standard Lisp I/O functions aren't as user friendly, but they do perform output and permit file I/O. In this section we'll see how these functions do console I/O. We'll discuss the specifics of file I/O in the next section.

read-line [file]	Returns a line of text entered at the keyboard or from a file.

When reading a line from the keyboard, read-line works quite similarly to (getstring t):

```
Command: (read-line)
This is a line of text
"This is a line of text"
```

`read-line` isn't as nice as `getstring` because it doesn't accept an optional prompt. The user won't know to enter text unless we issue our own prompt beforehand. For this reason, when we are seeking string input from the keyboard, we typically use `getstring`. `read-line` comes in handy when we specifically want to read a line of data from a file.

The `bp` function in Section 12.4 offers a situation where we want to use `read-line` for keyboard input.

`write-line` str {file}	Takes a string as its first argument and outputs that string to the screen or to a file.

`write-line` "pretty prints" the string, meaning that it removes the double quotes around it:

```
Command: (write-line "\nEnter your name")
Enter your name
"Enter your name"
```

The string is pretty printed once, then returned to the screen as a string because the `write-line` function is executed by the read-eval-print loop running at the Command: prompt. In fact, all of the output functions introduced in this section return a result to the screen because our examples are executed at the Command: prompt. When placed inside of a `defun`, the arguments are printed only once.

`write-line` is the companion to `read-line`; it's frequently used to write text to files. It is especially suited to this assignment because it automatically outputs a carriage return at the end of the line. Although `write-line` can also perform screen output, we'll shortly see some other I/O functions that are more appropriate for that task.

The strings that we provide `write-line` (and `princ` and `prompt` as well) can have *control characters* for output. The control characters are always preceded by a backslash and the characters themselves must be lowercase. They include the following:

\e	escape
\n	newline
\r	return
\t	tab
\nnn	the character whose octal ASCII code is nnn
\"	"
\\	\

The most frequently used control character is `\n`, which outputs a newline. This is equivalent to putting `<enter>` in the string.

`\t` inserts a tab and can be used to print data in columns, but it's not as useful as you might initially believe. The tab stops are an immutable eight characters apart, and if a datum is long, it can extend into the next column. We'll write a more flexible tabbing function as a lab in Chapter 9.

`\r` inserts a return without a newline, which causes subsequent text to overwrite the initial output. When would we use this option? Suppose we're running a calculation loop that requires many passes but doesn't display output on the screen. We'd like to print something so that the user knows that the function is still running.

`counter` is a function that prints a running counter on every pass through a loop. It overwrites the previous counter so that the output doesn't scroll off the screen. The `delay` function represents the calculation that we'd run the loop to perform. You can increase or decrease the numeric argument to `delay` if it runs too fast or slowly on your machine.

```
(defun counter (n / c)
  (terpri)
  (setq c 1)                          ;Initialize counter
  (repeat n                           ;Repeat requested # of times
    (princ "\rPass number: ")         ;\r allows overwriting
    (prin1 c)                         ;Print counter
    (command "delay" 20)              ;Wait a moment
    (setq c (1+ c)))                  ;Increment counter
(prin1))
```

Here's another use for control characters. Suppose we'd like to print

```
Length        2'3"
```

on the screen. We have a problem because the character we use to indicate inches, the double quote, is the exact same character we use to indicate the end of the string.

We can force a double quote into a string by preceding it with a backslash:

```
Command: (write-line "\nLength \t 2' 3\"")
Length 2' 3"                          ;printed
"\nLength \t 2' 3""                   ;returned
```

Notice that this example demonstrates the use of `\n` and `\t` as well.

How do we make a string contain a backslash? We use a double backslash:

```
Command: (write-line "c:\\acad\\support")
c:\acad\support
"c:\\acad\\support"
```

This is why we separate our DOS path components with a double backslash instead of a single one. Recall, that forward slash also works for this purpose.

`read-char` {file}	Reads a character entered at the keyboard or from a file and returns the decimal ASCII code for that character.

`read-char` is most often used to read a file character by character. Perhaps we'd like to read a comma-delimited database file and strip off the commas. In that case `read-char` is the tool to use.

`read-char` is not usually appropriate for obtaining keyboard input. Whenever it detects an end-of-line character (e.g., <enter> or <space>), `read-char` returns 10, which is the ASCII linefeed character. This can lead to lots of 10s being returned:

```
Command: (read-char)
a                                     ;User enters "a"
97                                    ;read-char returns its ASCII value

Command: (read-char)                  ;When read-char is run a second time
10                                    ; it automatically returns LF
```

Another function, `grread`, is better suited to reading a single character entered at the keyboard.

`write-char` int {file}	Takes the decimal ASCII code for a character as its first argument and outputs the character to the screen or to a file. It returns the ASCII code.

```
Command: (write-char 97)
a97
```

`write-char` is the companion to `read-char`. It's most often used to write a file character by character.

The `ascii` function can be used to convert a character to its ASCII equivalent.

`write-char` translates the LF character at the end of input to the appropriate end-of-line sequence for the platform on which it is running.

`terpri`	Outputs a carriage return to the screen and returns `nil`.

`terpri` does exactly what `(write-line "\n")` does, but if we only want to output a carriage return, it requires less typing.

`prompt` str	Outputs a string to the screen and returns `nil`.

As its name suggests, `prompt` is used to send a message to the user rather than output data. When we wish to report a condition or request information, we use `prompt`. Like `write-line`, `prompt` requires its argument to be a string and it pretty prints the message. It returns `nil` because the message is never part of our code.

```
Command: (prompt "\nAre you sure you wanted to format the c: drive? ")

Are you sure you wanted to format the c: drive? nil
```

`prompt` is the only output function that prints its argument on both screens in a dual-screen configuration.

`prin1` obj {file}	Takes an AutoLISP object as its first argument and outputs its printed representation to the screen or to a file. It returns the object.

`prin1` outputs an AutoLISP object as is; it makes no attempt to format it or to interpret any control characters embedded in a string.

We typically use `prin1` to output an AutoLISP object on the same line as something else we've just printed. An example is shown following `princ`.

`print` obj {file}	Takes an AutoLISP object as its first argument and outputs its printed representation to the screen or to a file. It precedes the output with a carriage return and follows the output with a space. It returns the object.

`print` is essentially equivalent to `terpri` followed by `prin1`. Since it does some formatting, `print` is the function we use most often to output AutoLISP data. An example is shown following `princ`.

`print` is also invaluable as a debugging tool. Its use in this capacity is explained in Section 12.4.

`princ` obj {file}	Takes an AutoLISP object as its first argument and outputs its printed representation to the screen or to a file. If the object is a string, `princ` "pretty prints" it. It returns the object.

For most AutoLISP objects, `princ` is identical to `prin1`. It differs only when its argument is a string. When it prints a string, `princ` strips off the double quotes and interprets any control characters embedded in that string. Like `prin1`, `princ` does not precede the output with a carriage return, and it returns the string as an AutoLISP object.

print, prin1, and princ are the standard Lisp I/O functions that we most often use to print AutoLISP data on the screen. They are only subtly different, as the following examples show. When you type these examples at the Command: prompt, conclude each by hitting the <space> bar instead of the <enter> key.

```
Command: (prin1 5) 55

Command: (print 5)                    ;precede with <enter>,
5 5                                   ; follow with <space>

Command: (princ 5) 55                 ;Identical to prin1

Command: (prin1 "\nstring") "\nstring""\nstring"

Command: (print "\nstring")
"\nstring" "\nstring"

Command: (princ "\nstring")           ;Pretty prints string
string"\nstring"
```

In these examples the arguments are printed twice solely because we typed them to the read-eval-print loop at the Command: prompt.

Both princ and prin1 may be called with no arguments at all, in which case they return nothing. This is actually quite useful when we create our own AutoCAD commands. As we know, built-in AutoCAD commands don't return nil or anything else when they're done; the Command: prompt is simply reprinted on the screen. We can have our own AutoCAD commands terminate in the same manner by making the last line be a princ or prin1 without any arguments.

Glance back for a moment at the c:long-dist command we wrote to illustrate the while function in Section 6.3. Near the bottom you'll find prin1s and princs used to output data, followed by a prin1 with no arguments. If you rerun the command, you'll see how it outputs its data and notice that it exits gracefully, without returning a result.

We use this exiting technique with AutoCAD commands only. We still want our AutoLISP functions to return results so they can be used by other AutoLISP functions or printed on the screen.

8.3 File I/O

Until now, all of the I/O we have seen has been at the user's console: out to the screen and in from the keyboard, mouse, or puck. It is frequently necessary for us to read and write data to files, thereby saving the data for future use.

All of the functions we saw in the previous section, except `terpri` and `prompt`, take an optional second argument that allows us to read from and write to files. However, file I/O in AutoLISP is fairly primitive. We must run functions to *open* the file before we start and *close* the file when we are finished.

`open` str str	Opens the file whose name is given as the first argument to perform the operation specified by the second argument. It returns a file descriptor.

The first argument to `open` is either a filename or a path to a file. We must remember to separate path components with a forward slash or a double backslash.

The second argument is one of the following three *lowercase* characters (if they aren't lowercase, `open` won't work!):

`"r"` Open a file to *read* its contents. If the file doesn't exist, `open` returns `nil`. If the file does exist, it is opened and can be read.

`"w"` Open a file to *write* new contents. If the file doesn't exist, a new file is created and opened. If the file already exists, its contents are immediately overwritten. Be careful!

`"a"` Open a file to *append* to it. If the file doesn't exist, a new file is created and opened. If the file already exists, data are appended to the end of it.

`open` returns a *file descriptor,* which is an AutoLISP object that we haven't yet seen. Its printed representation looks like this:

```
<File: #844d0334>
```

We use a file descriptor to access the file by making it the second argument to one of the standard I/O functions. But we never work with the file descriptor directly. Instead, we *always* issue the `open` function inside of a `setq` so that a

variable is bound to the file descriptor. We then supply that variable as the second operand to an I/O function. When the variable is evaluated, the file descriptor is returned. This is quite analogous to what we do with functions: We always specify symbols that name functions rather than providing the functions themselves.

On occasion we'll see a file descriptor printed on the screen, but we can't type one in ourselves; the AutoLISP reader won't recognize it.

An example should clarify this process. First let's open a new file to write (make sure you don't already have a file named `table.txt` before trying this!):

```
Command: (setq out (open "table.txt" "w"))
<File: #884d03a4>
```

Now the variable `out` is bound to the file descriptor. The file descriptor, in turn, is used to access `table.txt` as long as the file is open.

Let's write some data to this file:

```
Command: (write-line "Name \t Hours" out)
"Name \t Hours"

Command: (write-line "Joe \t 37.2" out)
"Joe \t 37.2"

Command: (write-line "Alexandria \t 43.7" out)
"Alexandria \t 43.7"
```

In these examples we supply the file descriptor as the second argument to `write-line`. Now each line of text is being written to `table.txt` rather than the screen. It's still being returned to the screen because we're at the Command: prompt, but unlike the examples shown in Section 8.2 it's only being printed once.

If we try to type the file at this point, nothing is printed. Run the DOS DIR command and the directory listing will show that the file has 0 bytes. Why is that? It's because the file isn't really there until we close it. So let's close the file:

```
Command: (setq out (close out))
nil
```

Now the file has been written to disk and the DIR command will find it.

`close` file	Takes a file descriptor, closes the specified file, and returns `nil`.

We *always* run `close` inside of a `setq` to reset the variable to `nil`. Otherwise the variable's value is still the file descriptor. What happens if we close the file without using `setq`? To learn that we must first reopen the file:

```
Command: (setq f (open "table.txt" "r"))
<File: #884d0396>
```

Notice that we've opened the file to *read.* If we reopen it to *write,* the file's contents will be destroyed. Now let's close the file and see what happens:

```
Command: (close f)
nil

Command: !f
<File: #884d0396>

Command: (type f)
FILE
```

The value of `f` is still the file descriptor, but it's a *dead file descriptor.* Since the file has been closed, the file descriptor is no longer usable:

```
Command: (read-line f)
error: file not open
```

Why does that matter? If we run a function that accesses our open files, it will think that `f` names an active file and will attempt to access it, thereby causing an error. Get in the habit of always putting `close` in a `setq`; it's elegant programming style and much safer.

Notes

- When our function requests a filename from the user, it should check whether the supplied file exists. An attempt to open a nonexistent file to *read* will lead to an error, whereas if our function opens an already existing file to *write,* the original contents will be destroyed. The easiest way to build in protection is to have the `getfiled` subr prompt the user for a filename.

- In general, leaving a file open for a long period of time is risky business. If a file is open and our machine crashes, we'll lose all the changes we've made

(as we've learned with our editor and drawing files!) and might even lose the file itself.

When we need to perform numerous tasks on the data in a file, it is safest to open the file, get the data, and close the file. Work with the data in memory, then reopen the file, write the data, and close the file again. Leaving a file open all day is an invitation to disaster.

- When initially writing a program, have it access data in memory. Once the program is completely debugged, add the code to perform file I/O. It can be very hard to debug a program that writes data to a file; if we make an error, the data might be sent into oblivion!

- AutoCAD automatically closes all of our files when we exit our drawing. If you wake up in the middle of the night and remember that you forgot to close an essential file, don't panic!

AutoLISP always writes ASCII text files. These files can be sent to the printer, typed to the screen, and modified with an editor. In DOS we can look at `table.txt` with the TYPE command (in Windows we must shell out to do this):

```
Command: type
File to list: table.txt
Name    Hours
Joe     37.2
Alexandria      43.7
```

Our data is nonaligned because the name "Alexandria" is very long and `\t` tabbed us over too far. We must plan for this possibility ahead of time or use a text editor to modify the file afterward.

Once a file has been created we can append data to the end. First, we must reopen the file `table.txt`:

```
Command: (setq out2 (open "table.txt" "a"))
<File: #884d0388>
```

This time we've chosen to make the file descriptor the value of the variable `out2`. It doesn't matter what symbol we use, although it helps clarify what we're doing if we give it a name that indicates its purpose. Many people use the symbol `f`, for *file,* as we did in a previous example.

Even though we're opening the same file as before, `open` returns a new file descriptor. Of course, since we work with a symbol whose value is the file descriptor rather than the file descriptor itself, this doesn't really concern us.

Now let's add some data to the file and close it:

```
Command: (write-line "Ingrid \t 66.5" out2)
"Ingrid \t 66.5"

Command: (setq out2 (close out2))
nil
```

When we look at the file, the new data are there:

```
Command: type
File to list: table.txt
Name     Hours
Joe      37.2
Alexandria     43.7
Ingrid   66.5
```

We can, of course, read the file:

```
Command: (setq in (open "table.txt" "r"))          ;Open file
<File: #884d037a>

Command: (read-line in)                            ;Read the first line
"Name \t Hours"

Command: (read-line in)                            ;Now read the second line
"Joe \t 37.2"

Command: (read-line in)                            ;Next, read the third line
"Alexandria \t 43.7"

Command: (read-line in)                            ;Yup, let's read the fourth line too!
"Ingrid \t 66.5"

Command: (read-line in)                            ;End of file
nil

Command: (setq in (close in))                      ;Close it
nil
```

Every time we issue a `read-line` function it returns the next line in the file. In this way we can sequentially read the file. We have no way of going backward through the file or accessing data randomly.

When `read-line` encounters and end-of-file, it returns `nil`. This means that `read-line` is an excellent test for an `if` or `while`, as in the following function:

```
(defun c:printit (/ filnam f lin)
  (setq filnam (getstring "Name of file to print: ")     ;Get filename
        f      (open filnam "r"))                        ; and open file
  (while (setq lin (read-line f))                        ;If more data, read it
    (write-line lin))                                    ; and print it
  (setq f (close f))                                     ;When no more data, close file
  (princ))                                               ; and exit gracefully
```

```
Command: printit
Name of file to print: table.txt
Name    Hours
Joe     37.2
Alexandria      43.7
Ingrid  66.5
```

The `while` loop prints each line of the file on the screen, then automatically exits when the end of file is encountered. This function enables Windows users to print a file without shelling out.

The version of `c:printit` in the `examples.lsp` file pauses following each screen of output, thereby allowing the user to read a large file. Try adding this capability to the preceding version of `c:printit` (it's not hard).

Note: Some text editors add a `Ctrl-Z` as an end-of-file marker. Any data written past this marker will not be input by `read-line`. Be certain to remove any `Ctrl-Z` that is not at the end of the file.

Example

The following example shows how we would write an AutoLISP object to a file, then read it back at a later time. In this case we'd use `print`, `prin1`, or `princ`; `write-line` only writes strings. The steps we perform are slightly different from the ones we've previously seen.

First, let's bind the variable `x` to a list:

```
Command: (setq x '(a b c d e))
(A B C D E)
```

Next, we'll open a file, print the list, and close the file:

```
Command: (setq o (open "temp.dat" "w"))
<File: #884d036c>

Command: (print x o)
(A B C D E)
```

```
Command: (setq o (close o))
nil
```

Now let's type this file on the screen:

```
Command: type
File to list: temp.dat

(A B C D E)
```

Notice that the first line of this file is blank. This is because we wrote the data to the file with the `print` function, and `print` precedes its output with a carriage return. When we read the file back in, we must remember that the first line is empty and discard the null string that is returned.

Now let's read this file back into memory:

```
Command: (setq i (open "temp.dat" "r"))
<File: #884d035e>

Command: (read-line i)                          ;Ignore the null string
""

Command: (setq y (read-line i))
"(A B C D E)"

Command: (setq i (close i))
nil
```

As we can see, the list does not come back in quite the same form that it went out:

```
Command: (type x)
LIST

Command: (type y)
STR
```

The reason is that any data we write to a file are stored as an ASCII string and `read-line` inputs the data as a string. Before we use the list, we must convert it back to an AutoLISP object with the `read` subr. If you are unfamiliar with this function, please study its contract in Section 9.2.1 before proceeding with this explanation.

From its name we might think that `read` is an input function, but it is strictly used for conversion. We use it with `read-line` to convert string input back to an AutoLISP object, like so:

```
Command: (setq z (read y))          ;Convert the list we read in
(A B C D E)

Command: (type z)                   ;Verify that it is a list
LIST
```

We generally combine the input and conversion steps into one:

```
Command: (setq y (read (read-line i)))
(A B C D E)
```

This is yet another programming cliché code that we use all the time.

By the way, do you think the two lists, x and z, are identical? Why?

```
Command: (eq x z)
?
```

When the read function converts the string into a list, it creates a brand new list; functions that create lists never reuse existing ones. Therefore the values of x and z are not eq.

8.3.1 The load Function Revisited

We use the load function to bring an AutoLISP file into memory so that we can run the commands and functions it contains. load has several facets that we have not yet seen.

The load function is introduced in Section 1.10.

load str [obj]	Takes the name of an AutoLISP file and loads that file into memory. It evaluates the second argument if the load fails.

As we know, if the file we attempt to load does not exist, load issues an error message. Let's try to load a nonexistent file:

```
Command: (load "bracket")
error: LOAD failed.
```

load takes an optional second operand, which is evaluated and returned when it cannot load the file. The operand can be any form we choose:

```
Command: (load "bracket" 'oops)
OOPS

Command: (load "bracket" (* 4 5))
20
```

When we supply AutoCAD commands to our coworkers, we want to make sure that the commands don't crash. Even if they are thoroughly debugged, there are some situations we can't control. For example, if we ask the user to enter the name of a file to load, we can't guarantee that the file exists or that the user spelled its name correctly.

An elegant program tests whether a file exists before attempting to load it. If it doesn't exist, the routine should prompt the user for a new filename. The code to accomplish the task might look like this:

```
(setq fname (getstring "\nEnter name of file to load: "))
(while (null (load fname nil))
  (setq fname (getstring "\nFile doesn't exist. Enter new filename: ")))
```

If the file doesn't exist, `load` returns `nil`. This causes `while`'s test to succeed, so it prompts for a new filename.

Another aspect of `load` is that it is recursive. If the file we load has a `load` function in it, then the specified file is loaded as well.

Let's say we're working on a project for which we've written a handful of AutoLISP routines. When the project is finished, we won't need these programs again. Rather than adding each of the functions individually to the `acad.lsp` file, we can insert a `load` function that pulls in another file containing the necessary routines. When the project is completed, we simply remove the `load` function from `acad.lsp`.

An additional feature of `load` concerns our file specification. If we don't supply a complete path when we load a file, the `load` function searches the AutoCAD library path for the specified file (see Section 8.4).

8.3.1.1 Automating the Edit-Load Cycle

By now we're quite familiar with the steps needed to create and run an AutoLISP routine. We create the file in the editor, load the file, then execute the function(s). During the debugging phase we might repeat the edit and load steps several (many?!) times.

The examples.lsp file contains a command named c:ed that greatly simplifies this process for anyone using a DOS-based editor. The command assumes that the AHED editor is in our current directory and that we've added AHED to the acad.pgp file. The code itself can be easily modified to use most DOS-based editors . . . just replace AHED with the name of your own editor.

Let's assume we'd like to edit a file named test.lsp. Load examples.lsp and enter the following code:

```
Command: ed
File to edit: test
```

This will call the AHED editor with the file test.lsp. The .lsp extension is assumed, and we *cannot* type it when we specify the filename.

When we exit the editor, c:ed prompts us to load the name of the file we just edited:

```
File to load <test>:
```

When we press the <enter> key, the file is automatically loaded for us.

The next time we issue c:ed during this drawing session, it prompts us with the name of the last file we edited:

```
Command: ed
File to edit <test>:
```

All we have to do is press <enter> and test.lsp is brought into the AHED editor once more.

We go through the edit-load cycle so many times that it's a tremendous time sink to continually issue these commands in their long form. Please add your editor to the acad.pgp file (if you haven't already done so) and experiment with c:ed. It really is easy to use, and the time we save by automating the edit-load process is alone worth the price of this book!

8.3.2 Replacing Standard AutoCAD Commands with Our Own Commands

We can modify the built-in AutoCAD commands through judicious use of the s::startup function. s::startup was created to circumvent an AutoLISP problem concerning the acad.lsp file. Recall that we put in acad.lsp those

functions that we always want to have present during our drawing sessions. As with any file, when `acad.lsp` is loaded, all of the forms contained therein are evaluated.

In the case of `defuns`, that means defining the functions. But we can also run any AutoLISP function as a script by putting it in the file *not* inside of a `defun`. When the file is loaded, this function is evaluated and executed.

It was discovered that `acad.lsp` is loaded early in the AutoCAD initialization process, before the AutoCAD drawing editor has been fully initialized. Certain AutoLISP functions that are intended to be run as a script don't seem to work. For example, the `command` function is not recognized.

We'll shortly present a useful example that demonstrates the restriction, then explain how to overcome it. But first we need to discuss an aspect of AutoCAD of which you may not be aware:

Did you know that in AutoCAD there really is no LINE command? Or CIRCLE or TEXT commands? The real names of these commands are .LINE, .CIRCLE and .TEXT. The word "LINE" is just an alias for the .LINE command.

We have the power to undefine any AutoCAD command by using the UNDEFINE command:

```
Command: undefine
Command name: line

Command: line
Unknown command. Type ? for a list of commands.
```

Try that on a colleague's machine! Although the LINE alias is undefined, we can still run the .LINE command:

```
Command: .line
From point:
```

This change is temporary; LINE is redefined the next time we open a drawing. We can even get LINE back during the current drawing session by running the REDEFINE command:

```
Command: redefine
Command name: line

Command: line
From point:
```

As AutoLISP programmers we can undefine any AutoCAD command that we don't like and replace it with one of our own. For example, we may wish to modify the END and QUIT commands to do some processing prior to exiting a drawing. Other AutoCAD commands can be customized for our specific needs as well.

Suppose we usually want to use the PLINE command rather than the LINE command, and we'd like to be warned whenever we try to use the latter. We can redefine the LINE command to do this by performing the following steps:

1. Create our own `c:line` command, such as the following:

```
(defun c:line (/ yn)
  (initget "Yes No")                              ;Limit user to Y or N
  (setq yn (getkword "\nDo you really want PLINE? <Yes>/No: "))  ;Yes is default
  (if (equal yn "No")                             ;If No,
      (command ".line")                           ; use LINE command
      (command "pline"))                          ;Else, use PLINE
  (prin1))
```

2. Undefined the existing LINE command as shown above. LINE must be undefined or AutoCAD won't use our version.

How does this example relate to the use of `s::startup`?

If we always want to replace the built-in LINE command with our own, we should put `c:line` in our `acad.lsp` file. We also want to put the UNDEFINE LINE command there, *not* in a `defun`, so that it will be run automatically. The problem is that at the time `acad.lsp` is loaded the AutoCAD environment has not evolved enough to undefine commands. We must place the UNDEFINE inside of an `s::startup`:

```
(defun s::startup ()
  (command "undefine" "line"))
```

The final initialization task AutoCAD performs before giving us control at the Command: prompt is to see whether we have defined a function named `s::startup`. If we have, it runs the function. This delays the evaluation of the forms in its body until the last possible instance and ensures their execution. Were it not for `s::startup` we would have to manually UNDEFINE commands at the start of a drawing session.

There is no limit to the amount of code we can put in the `s::startup` function. Typically we place there every function that we would like to have run as a script.

8.3.3 Obtaining Filenames via Dialog Boxes

Much of the information we used to type to AutoCAD can now be supplied by choosing selections from dialog boxes. AutoLISP provides several ways for our programs to get input through dialog boxes, one of which is `getfiled`.

`getfiled` str str str int	Displays a dialog box on the screen and prompts the user to choose a filename from a list. It returns the filename or the path to the file.

`getfiled` is an intricate function because it has numerous options. We can run it in its simplest form as follows:

```
Command: (getfiled "" "" "" 0)
```

The dialog box that `getfiled` puts on the screen has a different appearance in DOS than in Windows, but the functionality is similar. Figure 8-9 shows the Windows version:

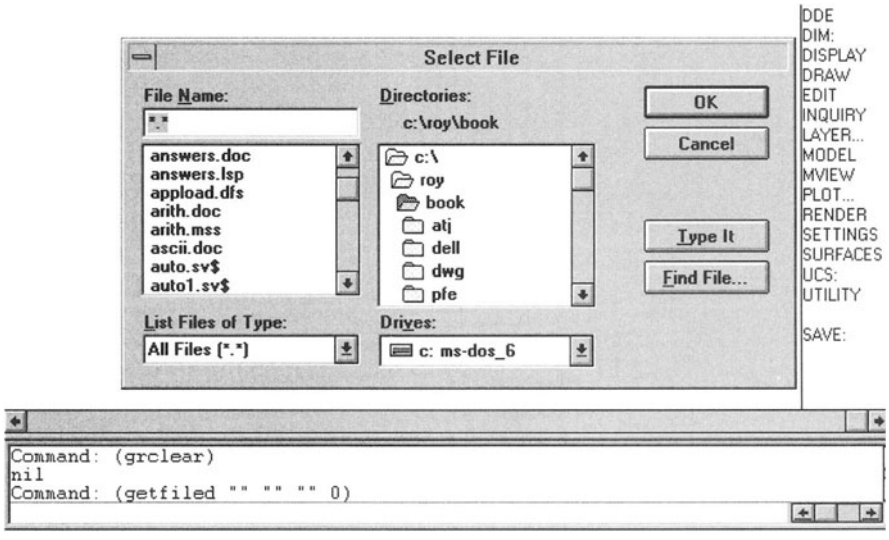


Figure 8-9

In this example `getfiled` brings up a dialog box that lists all of the files in the current directory. If the user types in a filename, double-clicks on a filename, or selects a file then clicks on the `OK` button, `getfiled` returns the complete path to the file. If the user clicks on the `Cancel` button, `getfiled` returns `nil`.

When we run the `getfiled` function, we may optionally provide a title, a filename or path, and an extension. If present, these become the default. The user then has the option of selecting the default or choosing a different file. Enter the following `getfiled`:

```
Command: (getfiled "Select a file to load" "examples" "lsp" 0)
```

Here is the dialog box that appears on the screen:

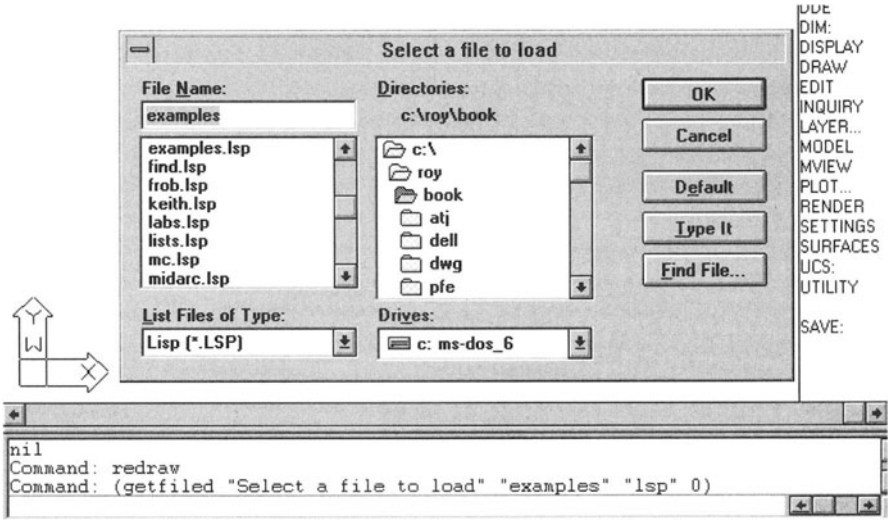


Figure 8-10

Now let's examine `getfiled`'s four arguments in more detail:

1. A string to be used as the title on the dialog box. If the null string is supplied, no title appears. The title in this example is `"Select a file to load"`.

2. A string specifying a default filename. If supplied, it's listed in the `File Name:` box (`File:` in DOS). If the null string is supplied, the `File Name:` box is left empty and the `Default` box is disabled. In this example, `examples` is the default filename.

   If the user clicks on a different filename in the file list, that name is entered into the `File Name:` box. She may also double-click on an entry in the `Directories:` list (including `..` and `\` in DOS) to see a listing of the files in a different directory. Unless it is disabled, the user can click on the `Default` box at any time to return to the filename that was initially supplied to the `getfiled` function.

If a path is supplied with the file, then the file list shows the files in that directory. Otherwise, the files in the current directory are displayed.

3. A string specifying the default extension. If supplied, the dialog box displays only the files with that extension. If the null string is supplied, then *all* files in the specified directory are displayed. If a `.dwg` extension is provided, `getfiled` previews the drawing in the dialog box. In this example, `.lsp` is the default extension.

4. An integer flag that controls the behavior of the dialog box. This flag is actually a bit setting that works exactly like the `initget` control bits; we can use them singly or add them together to effect multiple settings. When we don't want to set any flags, we supply 0 as we did in this example. If you're new to AutoLISP I suggest that you use 0 for now and come back to these other flags at a later time.

Four different flags can be set. Let's look at some examples of how they are used.

- We set a bit value of 1 when we want the user to enter the name of a file to be *created.* If the user selects an already-existing file, `getfiled` displays an alert box that warns the user and asks whether she wants to replace the file.

```
Command: (getfiled "Specify a new drawing file" "dwg/" "dwg" 1)
```

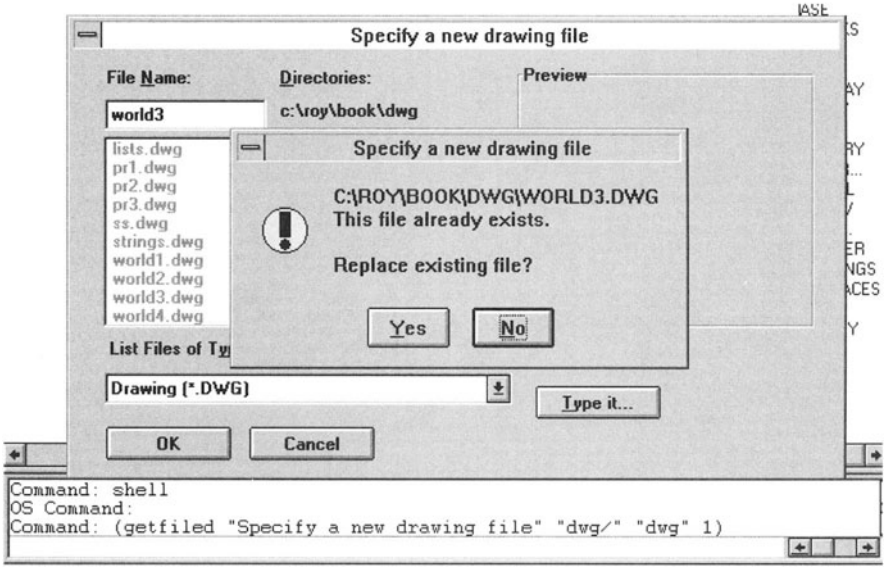


Figure 8-11

This example displays all the .dwg files in the d:\roy\book\dwg directory. If the user types a new filename, the complete path is returned. If she selects or types the name of an existing file in this directory, a warning is issued.

- Suppose the user wants to supply a filename to getfiled but doesn't remember where on the path the file is located. She can move around the directory structure until she locates it, but that could be very time-consuming. An alternative approach would be to click on the Type it button, which causes getfiled to return a result of 1 rather than a file path string. This would alert our program to use the findfile subr to locate the file in the following manner:

```
(if (numberp (setq f (getfiled "Select file to load" "" "" 0)))
    (setq f (findfile (getstring "Enter filename: "))))
```

A getfiled bit value of 2 disables the Type it button. It is set if another dialog box is active when getfiled is called. We would set this bit if we want the user to select an existing filename with her pointing device instead of typing it in (e.g., when we *don't* have the preceding code in our program). However, even if we set this bit value, she is still able to type in a filename.

- If we supply an extension as the third argument to getfiled, only files with that extension are displayed in the file list. Furthermore, the user is constrained to select a file with that extension. We set bit value 4 to eliminate this constraint and permit the user to select a file with a different extension.

  For example, if we had set bit value 4 in Figure 8-11, the user would be allowed to enter a filename with an extension other than .dwg.

- getfiled normally returns a complete path to a file. If bit value 8 is set and bit value 1 is not set, getfiled only returns the filename and extension. Furthermore, if we supply a filename and extension, getfiled searches the AutoCAD library path (see Section 8.4) to locate the file.

```
Command: (getfiled "Choose a LISP file" "filter" "lsp" 8)
"FILTER.LSP"
```

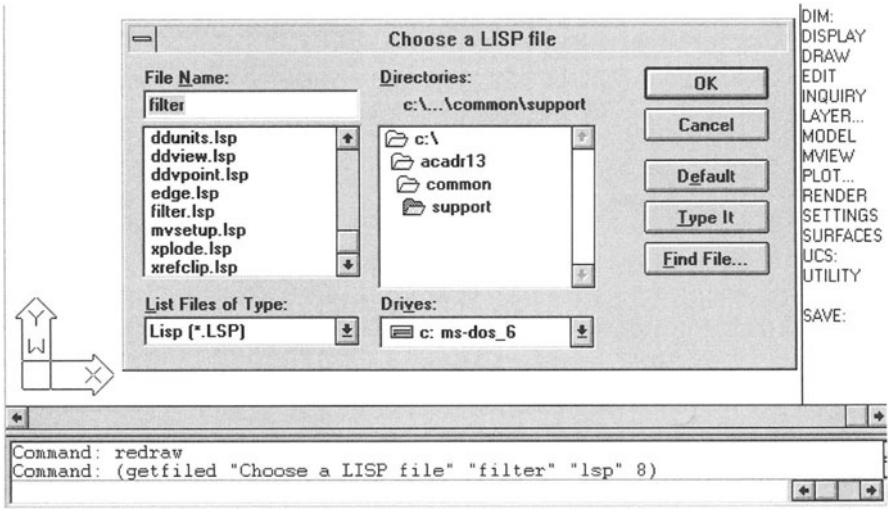


Figure 8-12

`getfiled` finds `filter.lsp` in the `"c:\acad\support"` directory and lists the files in that directory. Assuming that the user accepts the default, `getfiled` returns the string `"FILTER.LSP"`.

This option is especially useful when we want to locate and save a filename for later use, but we expect that the file will be placed in different directories on different machines. In this case we may want the filename only, not the complete path.

If bit value 1 is set, `getfiled` ignores bit value 8. It only lists files in the current directory, and it returns the full path to the selected file.

Note: During the investigation of `getfiled`, several bugs were discovered. See the `readme.txt` file on the enclosed floppy for the latest status of this function.

8.4 Advanced File Access

The functions `findfile` and `getenv` allow us to write more general code for accessing files and variables. These functions frequently use *system environment variables*, which are variables that we set at the DOS prompt prior to entering AutoCAD.

Those who have been using AutoCAD for a while may be familiar with the system environment variables `acadfreeram`, `lispstack`, and `lispheap`. In the old days we would set these variables to control the allocation of memory for AutoCAD. These are now done automatically for us.

We can set our own system environment variables for a variety of purposes. In particular, there is one variable that AutoCAD uses quite frequently, named `acad`. We typically set this variable to one or more directory paths that contain files we need to access. For example, the following command sets the `acad` system environment variable to the path `c:\connect` (the DOS prompt is shown as well):

```
>set acad=c:\connect
```

Notes

- UNIX is case sensitive; on that operating system we must call this variable `ACAD`, not `acad`.

- The DOS SET command must be executed prior to entering AutoCAD. We cannot push to DOS from AutoCAD to run it. System environment variables that we set at DOS level tend to disappear. If we want permanence, we should set them in the `autoexec.bat` file.

- We can create and give values to our own system environment variables in a similar manner, then access them from within AutoCAD and AutoLISP. For example, we might need to know the path to a prototype drawing or a file to be inserted. In AutoLISP we access system environment variables directly by `getenv` and indirectly by `findfile` and `load`. These functions are described below.

- The size of our system environment space limits the number of system environment variables we can have. We can increase this by giving a larger number to the SHELL command in the `config.sys` file.

The directory specified by the `acad` system environment variable is part of the *AutoCAD library path.* When AutoCAD searches for a file, it does *not* search the path specified in our `autoexec.bat` file. Instead it scans this library path, in the following order, and returns the first occurrence of the file that it finds:

1. The current directory. This is defined to be the directory that was current when AutoCAD was started, such as `c:\machine`. If we shell out and change directories to, say, `c:\project`, `c:\machine` is still the current directory.

2. The directory containing the current drawing file (if different from our current directory), such as `c:\machine\assembly`.

3. The directory named by the `acad` system environment variable. We've chosen `c:\connect` for our example, but we can specify as many paths as we want, separated by semicolons:

```
>set acad=c:\connect;c:\machine;c:\machine\assembly
```

4. The directories containing the AutoCAD program files (i.e., the AutoCAD system directories). This might be `c:\acad` or `c:\acadr13` and its subdirectories.

The examples in this section assume that the library path locates the directories shown above.

When we run a `load` function in AutoLISP, it searches the AutoCAD library path. If the file we seek is in our current directory or another directory along the path, the file will be found. But if we work in a directory that's different from the one that was current when we began our AutoCAD session, `load` won't locate the file. This problem has several solutions:

- Always specify the complete path to the file. If we're frequently loading files from the Command: prompt, this process is very tedious and time-consuming.
- Initially invoke AutoCAD from the directory we plan to work in. This is the easiest solution, but some installations require AutoCAD to be started via a batch file that automatically makes a specific directory current.
- Set the `acad` system environment variable to the directory in which we plan to work. Then functions that search the library path will be able to find our files.

Unlike `load`, `open` doesn't use the AutoCAD library path. If we open a file to read or append and that file is not in our current directory, `open` won't find it. We can compensate for this limitation with the `findfile` function.

`findfile` str	Searches the AutoCAD library path for the supplied filename and returns the path to the file, or `nil` if the file is not found.

We use `findfile` in the following manner:

```
Command: (findfile "flange.lsp")
"c:\\connect\\flange.lsp"
```

`findfile` locates `flange.lsp` by searching the AutoCAD library path. It checks the current directory, `c:\machine`, then the directory containing our drawing file, `c:\machine\assembly`, before finding the file in the `connect` directory, which was specified by the `acad` system environment variable. Since it found the file, it doesn't need to search the directories containing the AutoCAD program files.

`findfile` is particularly useful when we wish to open a file for I/O. As we have just seen, we can access files in directories other than our working directory even if we don't know the exact path:

```
Command: (setq f (open (findfile "flange.lsp") "r"))
<File: #884d0350>
```

This is especially handy when different users are on different network nodes and we must make our code drive-independent.

We can also use `findfile` to learn whether a file exists before we try opening it. Remember, if we open an existing file to write, its contents will be destroyed. The following code is a safety measure:

```
(setq f (getstring "Name of file to write: "))
(while (findfile f)
  (setq f (getstring "File already exists; choose another name: ")))
```

A problem with this code is that it may return the name of a file in another directory if that directory is on the AutoCAD library path. Another test would be to open the file to read; if the file doesn't exist, `open` returns `nil`. However, if the file does exist, we must remember to close it. This test is quite time-consuming because file I/O is comparatively slow. Probably the best algorithm is to use the `getfiled` function whenever it's appropriate.

Note that, unlike `load`, `findfile` makes no assumptions about the file's extension. When we want to access an AutoLISP file, the `.lsp` extension is required.

`getenv str`	Takes the name of a system environment variable and returns its value.

The `getenv` function provides AutoLISP access to the system environment variables. Its argument must be a string, and it returns a string as well. If we expect the value of the system environment variable to be a number, we can use the `read` function to convert it.

We can use getenv to learn the value of the acad system environment variable in the following manner:

```
Command: (getenv "acad")
"c:\\connect"
```

getenv can also be used to access system environment variables that we create. This allows us to specify directories that are not on the AutoCAD library path and to access them at runtime.

For example, let's load the file wing-nut.lsp. On our system it is stored in the directory c:\connect\hardware. But it might be in a different directory on another workstation. We can designate the directory at the DOS prompt in the following manner:

```
>set parts=c:\connect\hardware
```

Inside of AutoCAD we can access this path with getenv:

```
Command: (getenv "parts")
"c:\\connect\\hardware"
```

What does this gain us? Now we can load wing-nut.lsp from the directory specified by the parts system environment variable:

```
Command: (load (strcat (getenv "parts") "/wing-nut"))
```

The strcat function concatenates or joins strings together. In this example it joins the value of the parts system environment variable, "c:\\connect\\hardware", to the string "/wing-nut", giving the complete pathname "c:\\connect\\hardware\\wing-nut.lsp".

In other words, we are able to load the file wing-nut.lsp from the directory specified by the parts system environment variable rather than from a specific, hard-coded directory. This gives us machine independence; we can use the same code on any machine by making the appropriate path be the value of the parts system environment variable.

Example

getenv can be used in conjunction with findfile to locate a file in a machine-independent manner. Assume we keep track of our small and large hardware inventory in the files small.dat and large.dat, which are generally located in the directory named by the parts system environment variable.

It's good programming practice to test whether one of these files is present in that directory before opening it, in order to avoid an error.

Assume that the `parts` system environment variable again names the path `"c:\connect\hardware"` and that `large.dat` is *not* in this directory. We can run the `c:get-parts` command as follows:

```
Command: get-parts
Which parts (Small or Large)? l
c:\connect\hardware\Large.dat wasn't found
```

Since `c:get-parts` can't find the file, it prints an error message rather than crashing. In practice we'd probably use a `while` loop to reprompt the user instead of simply exiting. Let's examine the code.

```
  (defun c:get-parts ( / file f in)
1   (initget 1 "Small Large")                          ;User's choices (can't hit enter)
    (setq file (getkword "Which parts (Small or Large)? "))
2   (setq file (strcat (getenv "parts") "\\" file ".dat"))   ;Build path
3   (if (null (setq f (findfile file)))                ;Does file exist?
        (prompt (strcat "\n" file " wasn't found"))    ;No, print error message
        (progn                                         ;Otherwise,
          (setq in (open f "r"))                       ;  open the file
          (princ (read-line in))                       ;  read its contents
          (setq in (close in))))                       ;  and close it
    (prin1))
```

1. `initget` limits input to the string `"Small"` or `"Large"`, the user's not even allowed to press the <enter> key. `getkword` prompts the user to enter one of those strings and assigns it to the variable file.

2. The directory string specified by the `parts` system environment variable is concatenated to the filename selected by the user and the `.dat` extension, building a complete file path.

3. If the file isn't present in the directory, `findfile` returns `nil`, the `if` test succeeds, and the error message is printed. If the file does exist, it is opened, its contents are read, and it's closed once again.

8.5 Important Points to Remember

- Making the last line of an AutoCAD *command* be `(princ)` or `(prin1)` allows the command to exit gracefully. In general, we don't end our AutoLISP *functions* with `princ` or `prin1`.

- Always run `open` inside of a `setq`; we must bind a variable to the file descriptor so that we can access it. Always run `close` inside of a `setq` as well. Otherwise our symbol remains bound to the file descriptor and we (or a program) might think that the file is still open.

- Keep files open for as short a time as possible. If the machine crashes when a file is open, we will lose the changes we made and might even lose the entire file.

- When initially writing a program, have it access data from memory. Add the code to perform file I/O once the program has been debugged.

- When we write an AutoLISP object to a file, it is stored as a string. When we read in that string, we must convert it back into an AutoLISP object. For example, if the value of the symbol `f` is a file descriptor, enter

```
(read (read-line f))
```

  to input one line of the file and convert it into an AutoLISP object.

8.6 Labs

1. Convert the `grades` function we wrote in Chapter 6, Exercise 6 into a command. Disallow negative, 0, and null input.

```
Command: grades
What is your grade? -22

Value must be positive and nonzero.
What is your grade? 94
EXCELLENT
```

2. In Chapter 6, Lab Exercise 17 we created a function that used a `cond` to perform various zooms. Transform this function into a command that prompts the user for the allowable strings and repeats until the user presses `<enter>`.

3. Write a predicate called `y-np` that takes a string as its only argument, prints that string on the user's terminal, and prompts for a response of `"yes"` or `"no"`. Use `getkword` and return `t` or `nil`, depending on the user's response.

```
Command: (y-np "Beam wide enough?")
Beam wide enough? (Yes or <No>) maybe

Invalid option keyword.
Beam wide enough? (Yes or <No>) y
T
```

In this example `"No"` is the default; make `"Yes"` the default if you desire. Note that you may need the `strcat` function, which is described in the next chapter.

4. Rename the `c:endit` command that we wrote in Chapter 5, Lab Exercise 12 to `c:end` and create an `s::startup` function that undefines the AutoLISP END command. Add both to your `acad.lsp` file so that the END command saves your drawings in a smaller space.

5. Write a function called `otos` that takes an AutoLISP object and converts it to a string. To do this we must write the object to a file then read it back in.

```
Command: (otos 33)
"33"

Command: (otos "abc")
"abc"
```

A Note Concerning Lab Exercises 6 to 8

This book endeavors to present labs that solidify your understanding of AutoLISP functions, concepts, and techniques. Although I have tried to make the exercises useful, my main objective is for each lab to give you practice with a specific new AutoLISP subr.

The way to gain true comfort with AutoLISP is to write a useful project that incorporates different programming tools and techniques. The remaining three exercises constitute such a project.

The entire project entails only a page of code, yet it requires you to use lists and association lists; local and global variables; conditionals and loops; terminal I/O and file I/O; user interface techniques. In short, it incorporates all of the AutoLISP concepts we've seen thus far. Once you've completed this project, you'll feel much more comfortable writing in AutoLISP.

6. Write a command that repeatedly prompts the user for a part number and a part name until some end condition is met (e.g., user presses `<enter>` instead of supplying useful data). The part number should be either an inte-

ger or a real number, and the part name should be either a string or a symbol (whatever is more practical for your imagined application). Cons each number/name pair onto an association list, as follows:

```
Command: parts
Part number: 1
Part name: door
Part number: 2
Part name: window
Part number:

((2. WINDOW) (1. DOOR))
```

By consing pairs onto the association list, our final list is in reverse order of the way the pairs are entered. Since we don't care about the order, *don't* reverse the list when done.

For now, make the association list be the value of a global variable so that the next function we write can access it. Later, when we add file I/O, we can make this variable local.

7. Write a command that uses the association list you created in the last exercise. Repeatedly prompt the user for a part name until some end condition is met. Run the `assoc2` function (Chapter 7, Lab Exercise 11) on each part name, and print its associated part number.

```
Command: get-nums
Part name: door
The part number for door is 1

Part name: window
The part number for window is 2

Part name:
```

8. Modify the `parts` command to prompt the user for the name of a data file (default the extension to `.dat`), then write the association list to that file. Next, modify `get-nums` to prompt for a filename (using `getfiled`) and read in the association list from that file. Replace the global variables used in the previous two exercises with local variables.

   Extra credit: Have `parts` reprompt the user if the supplied filename already exists.

   Note: Make sure Lab Exercises 6 and 7 run correctly before adding file I/O. It is much easier to debug a program when the data are in memory than when they are on the hard disk.

9

Expanding our AutoLISP Toolbox

Topics Covered in This Chapter

- String manipulation functions and wildcards
- String, number, and angle conversion
- Functions for performing measurements
- Symbol manipulation functions
- Functions that manipulate the graphics screen or images on the screen
- The AutoCAD Development System

Goals for This Chapter

- Understand how strings are used and manipulated.
- Learn how to convert an object to a different datatype.
- Learn how to perform AutoCAD-like measurements in AutoLISP.
- See ways to control the presentation of the graphics screen.
- Learn the contracts of several additional subrs.

Introduction

This chapter explains the contracts of numerous AutoLISP functions. By now we're quite familiar with how AutoLISP works, and it's time for us to add subrs to our toolbox.

Depending on your applications, you'll use some of these functions almost daily and others not at all. But everyone's needs differ. It is very worthwhile having a basic understanding of what all of these functions do so that when they are needed you'll know they exist. You can study their contracts in more detail at that time.

9.1 String Functions

Strings are employed in a variety of ways. We have seen that they are useful as prompts and that filenames and paths to files are expressed as strings. We also hand strings to the TEXT command when we want to place them in our drawings. We use four AutoLISP functions to manipulate strings.

`strcat` str . . . str	Returns a new string consisting of all of the characters in the supplied strings.

`strcat` is short for ***STR***ing con***CAT***enate; *concatenate* means to "join together." `strcat` does to strings what `append` does to lists. In fact, in Common Lisp this function is called `string-append`. The following examples show how string concatenation is done:

```
Command: (setq x "22")
"22"

Command: (setq y (strcat x ".46"))
"22.46"

Command: (strcat "The bar is " y "\" long.")          ;\" puts a double
"The bar is 22.46" long."                             ;quote into a string
```

Example

The `c:ed-title` command asks the user to choose an area of the title block to modify, then returns that choice. On the first pass, the default choice is `"eXit"`; thereafter the default becomes whatever the user chose on the previous invocation of the command. Here's how we'd run it:

```
Command: ed-title
Section to change (DAte/DWgtitle/Jobtitle/Chk/Rev/eXit <eXit>): r
"Rev"

Command: ed-title
Section to change (DAte/DWgtitle/Jobtitle/Chk/Rev/eXit <Rev>): d
Ambiguous response, please clarify . . .
DAte or DWgtitle? da
"DAte"
```

ed-title saves its result in the variable s#. s# must be a global variable because we'll need its value the next time the command is run. Were s# local, it would disappear at the conclusion of the command. We give it an unusual name so that it's not accidentally rebound by another function. Let's examine the code:

```
  (defun c:ed-title (/ temp)
1    (if (null s#)                              ;If first time
        (setq s# "eXit"))                       ; set default prompt to "eXit"
2 (initget "DAte DWgtitle Jobtitle Chk Rev eXit")
  (setq temp (getkword                          ;Display choices
    (strcat "Section to change (DAte/DWgtitle/Jobtitle/Chk/Rev/eXit <" s# ">): ")))
3 (if temp                                      ;User didn't hit enter (temp not nil)
     (setq s# temp))                            ; so assign user's choice to s#
  s#)                                           ;Return s# as result
```

1. The first time this command is executed the value of s# is nil, so it is set to the default, "eXit".

2. The user's input is limited by initget and the acceptable choices are displayed by getkword. getkword's prompt consists of three components:

   - The string
     "Section to change (DAte/DWgtitle/Jobtitle/Chk/ Rev/eXit <"
   - The value of the variable s# (which is a string)
   - The string ">) : "

   We can't make the prompt one simple string because s# must be evaluated. Therefore, we use strcat to build the prompt by concatenating the three strings together. This is a classic application of strcat.

3. If the user presses <enter> in response to the prompt (thereby accepting the default), getkword returns nil. If the user supplies a keyword, s# is rebound to that string for the next time c:ed-title is called. The value of s# is returned as the result of running the command.

strlen str	Returns the number of characters in a string.

```
Command: (strlen "short string")
12
```

There are twelve characters in the string `"short string"`. Note that spaces are valid characters in a string and are counted.

`strlen` does for strings what the `length` function does for lists. It can be used in conjunction with `repeat` when we need to access each character in a string:

```
(repeat (strlen string)
  .
  .
  .)
```

strcase str {t}	Returns a copy of the supplied string with all alphabetic characters converted to uppercase. If a second argument of `t` is supplied, the string is converted to lowercase.

```
Command: (setq p "Center")
"Center"

Command: (strcase p)
"CENTER"

Command: (strcase p t)
"center"
```

Uppercase and lowercase strings are not the same. When we test a user's input, we must make certain that we compare same-case strings. One way to accomplish this is with `strcase`:

```
(setq s (getstring "Enter entity to modify: ")
(if (equal (strcase s) "LINE")
    .
    .
    .)
```

This code converts the user's input to uppercase to test whether it matches the uppercase string `"LINE"`.

`substr` str int {int}	Takes a string and a starting character position and returns a portion of the string (a substring) beginning at that position.

In the following example the second argument is 3, so `substr` returns the remainder of the string beginning with the third character:

```
Command: (substr "This is a string" 3)
"is is a string"
```

`substr` takes an optional third argument, which is the number of characters we want returned:

```
Command: (substr "This is a string" 3 7)
"is is a"
```

This example returns a substring beginning with character position 3 and totaling 7 characters. Note that spaces are counted as characters in strings.

Example

The following useful example parses a long string into a list of string tokens. For example:

```
Command: sparse
Input string: Lower window 3 inches
("Lower" "window" "3" "inches")
```

This function is quite handy when inputting data from files. Remember that AutoLISP data is always stored in files as ASCII and read back as strings. We can use the `read` function to convert the strings back into AutoLISP data.

One limitation of `read` is that if we give it a multiword string, it only converts the first word. We can overcome this limitation by parsing the long string into a list of individual strings with `c:sparse`, then running `read` on each element of the list using `foreach`. In section 9.2.1 we'll see a function called `convstrs` that accomplishes this task.

`c:sparse` is based on a command by Bill Kramer.

```
1 (defun c:sparse (/ s char cnt str-list word)
    (setq word ""
          cnt  0
          s    (getstring t "\nInput string: "))
2   (while (< cnt (strlen s))                          ;If not end of string
      (setq cnt  (1+ cnt)                              ; increment cnt and
            char (substr s cnt 1))                     ; get next character
3     (if (eq char " ")                                ;End of word?
        (if (not (eq word ""))                         ;Yes. If not a null string
            (setq str-list (cons word str-list)        ; cons it onto str-list
                  word " "))                           ; and reset word
        (setq word (strcat word char)))                ;No, add char to end of word
    ) ;end of while
4   (if (eq word "")                                   ;Any remaining characters?
        (reverse str-list)                             ;No, reverse and return list
        (reverse (cons word str-list))))               ;Yes, cons on word first
```

1. Step 1 of `c:sparse` is initialization. It uses the following local variables:

`s`	Its value is the string the user inputs.
`char`	On each pass through the loop its value becomes the next character in the string.
`cnt`	It is bound to the current character position and used by `substr` to access the next character. Initially 0, it's incremented during each pass through the loop.
`str-list`	Initially `nil`, its value becomes the list of parsed words.
`word`	Initially the null string, it's used to build a specific string token. Once a token is complete, it is consed onto the front of `str-list`, and `word` is reset to the null string.

2. The `while` loop is repeated once for every character in the string. On each pass `cnt` is incremented by 1, and the loop is repeated until `cnt` equals the length of the string. `char` is set to the next character in the string by taking a substring of `s` beginning with the character position indicated by `cnt`, for one character.

3. This is the core of the function. If the value of `char` is *not* a space, the ELSE path is taken, and the current character is concatenated onto the end of `word`.

   If the value of `char` *is* a space (`" "`), then the value of `word` should be an entire word. However, if the initial string contains several consecutive spaces, `char`'s value could conceivably be a space while `word`'s value is still the null string. The inner `if` tests for this condition. If the test fails, then `word`'s

value is *not* the null string, in which case the token is consed onto the front of `str-list` and `word` is reset to the null string.

4. When the `while` loop is finished, `c:sparse` tests whether the value of `word` is the null string. If it's not, the final token is consed onto the front of `str-list`. `str-list` is then reversed (thereby putting the words in the correct order) and returned as the result.

9.1.1 Wildcards

The wildcard facility adds tremendous flexibility to our use of strings. Wildcards are familiar to most of us because we often use the `*` wildcard to match filenames in DOS or UNIX. For example, in DOS we might enter

```
>dir *.lsp
```

to obtain a directory of all our Lisp files. Wildcards work similarly in AutoLISP, but they're much more powerful and versatile than in DOS.

A predicate called `wcmatch` works explicitly with wildcards (in fact, its name stands for ***W***ild***C***ard ***MATCH***) and demonstrates how wildcards work.

`wcmatch` str str	Compares two strings, the second of which is a wildcard pattern, and returns `t` if the first string matches the pattern.

The following wildcard characters are used by `wcmatch`, as well as by other AutoLISP functions and AutoCAD commands:

`#` Matches any single numeric digit.

```
Command: (wcmatch "bar4" "bar#")
T
```

`@` Matches any single alphabetic character.

```
Command: (wcmatch "bar4" "bar@")          ;4 isn't alphabetic
nil

Command: (wcmatch "bara" "bar@")
T
```

. Period matches any single nonalphanumeric character.

```
Command: (wcmatch "bar-7" "bar.7")          ;hyphen is nonalphanumeric
T
```

? Period matches any single character whatsoever.

```
Command: (wcmatch "layer3" "layer?")
T

Command: (wcmatch "layer3" "layer??")       ;Nothing to match 2nd ?
nil
```

* Matches any group of zero or more characters.

```
Command: (wcmatch "foo.lsp" "*.*")
T

Command: (wcmatch "abc" "*a*")
T
```

~ If it's the first character in the pattern, then it matches anything that is *not* the pattern.

```
Command: (wcmatch "abc" "~abc")             ;"abc" matches "abc"
nil                                         ;  so test fails

Command: (wcmatch "a~c" "a~c")              ;Not the first character
T                                           ;  so not a wildcard char.
```

[-] Used to specify a range of characters to match.

```
Command: (wcmatch "test-M" "test-[A-Z]")
T

Command: (wcmatch "test-M" "test-[a-z]")    ;"M" is not in range "a-z"
nil

Command: (wcmatch "str38" "str[1-38]")      ;Matches "str1" - "str3", or
"str8"
nil                                         ;Doesn't match "str38"
```

~ [-] Matches anything not in the range.

```
Command: (wcmatch "4" "~[0-3]")
T
```

, Comma is used to specify more than one pattern to match against. It succeeds if the string matches any of the patterns.

```
Command: (wcmatch "layerX" "layer[1-8],layer[A-Z]")
T        ;"layerX" matches one of these patterns
```

` Back quote takes the following wildcard character literally. In other words, it does for wildcards what backslash does for control characters in strings. In the following example, the backquote indicates that the brackets are not wildcard characters, they are part of the string.

```
Command: (wcmatch "[a]" "`[a`]")
T
```

The *AutoLISP Reference* suggests backquoting all nonalphanumeric characters in a wildcard pattern in case future AutoCAD releases use one of them as a wildcard character.

Wildcards can also be used with most AutoCAD commands that list symbol names. The following example requests a listing of the information for Layers 0, 1, and 2:

```
Command: layer
?/Make/Set/New/ON/OFF/Color/Ltype/Freeze/Thaw: ?
Layer name(s) to list <*>: [0-2]

Layer name   State     Color        Linetype
-----------  -------  ------------  ----------
0            On       7 (white)     CONTINUOUS
1            On       3 (green)     CONTINUOUS
2            On       2 (yellow)    CONTINUOUS
```

Wildcards are especially useful in this regard when we want to freeze, say, every layer except Layer 2.

9.2 Conversion Functions

By now we're well aware that each function requests certain types of data to work with. We frequently find that the data we want to use are not in the correct format for the function we want to use them with. For example, we may want to put a number in our drawing with the TEXT command, but TEXT requires a string. We must therefore convert the number into a string before we give it to the TEXT command. The functions in this section are used to perform various types of conversions.

The conversion functions never change the data we give them. Instead they return a copy of the data altered in an appropriate manner. For that reason we often run a conversion function inside of a `setq` or within another function that will use the data immediately.

A couple of conversion functions that we use quite frequently are `dtr` and `rtd`. These functions convert ***D***egrees ***T***o ***R***adians and ***R***adians ***T***o ***D***egrees, respectively. Although widely used in our programs and throughout this book, neither function is a subr; we wrote them in Chapter 5. These functions are prime candidates for our `acad.lsp` file.

9.2.1 String Conversions

`atoi str`	Converts an ASCII string into an integer.

```
Command: (atoi "97")
97

Command: (atoi "3.8")
3
```

If we give `atoi` a string representing a real number, it converts it to an integer and truncates its fractional component. If we want to keep it a real number, we must use `atof`.

In the case of strings such as these, `atoi` works very similarly to `read`. The difference is that `read` won't fix the number. We use read for the more general case and use `atoi` when we want to specifically convert an integer.

atof str	Converts an ASCII string into a floating point (i.e., real) number.

```
Command: (atof "3.4")
3.4

Command: (atof "22")                    ;atof floats integers
22.0
```

ascii str	Converts the first character in a string into its numeric ASCII equivalent.

```
Command: (ascii "A")
65

Command: (ascii "abc")
97
```

chr int	Takes an ASCII character code and returns its character equivalent.

```
Command: (chr 65)
"A"
```

Appendix B contains a listing of ASCII codes.

read str	Takes a string and returns the AutoLISP object that the string represents.

read is a conversion function whose actions are demonstrated by the following examples:

```
Command: (setq a (read "Channel"))      ;Convert to symbol
CHANNEL

Command: (type a)
SYM

Command: (setq b (read "(x y z)"))      ;Convert to list
(x y z)
```

```
Command: (type b)
LIST

Command: (+ (read "43") 7)                    ;Convert to number and add 7
50
```

If the string contains a space, everything up to the space is converted and everything after the space is ignored:

```
Command: (read "Tuesday afternoon")
TUESDAY
```

In Section 9.1 we used the `c:sparse` command to parse the multiword string, `"Lower window 3 inches"`, into a list of smaller strings, `("Lower" "window" "3" "inches")`. The following function takes such a list and uses `read` to convert the strings into the AutoLISP objects that they represent:

```
(defun conv-strs (str-list / obj-list)
  (foreach str (reverse str-list)                    ;Access each string in turn
    (setq obj-list (cons (read str) obj-list))))     ;Convert and cons it onto obj-list

Command: (conv-strs '("Lower" "window" "3" "inches"))
(LOWER WINDOW 3 INCHES)
```

From its name we might think that `read` is an input function, but it is strictly used for conversion. Perhaps it was named "read" because it's frequently used with `read-line` to convert string input back to an AutoLISP object. For example, if the symbol `f` is bound to a file descriptor, we would input AutoLISP data from the file by entering `(read (read-line f))`. Refer back to Section 8.3 for a more complete example showing the use of `read` with file I/O.

♦ ♦ ♦

Occasionally we want to convert a symbol into a string, but AutoLISP has no subr to accomplish this task. Recall, however, that any data written to a file are stored as an ASCII string and read back in as such. Although it's a slow and roundabout method, when we need to convert a symbol into a string we write that symbol to a file and read it right back in. The `otos` function that we wrote in Chapter 8, Lab Exercise 5 does this for us.

9.2.2 Number Conversions

`itoa` int	Takes an integer and converts it into an ASCII string.

```
Command: (itoa -33)
"-33"
```

`itoa` performs the reverse conversion of `atoi`.

`rtos` num {int} {int}	Takes a real number and converts it into a string. It accepts an optional editing mode and a precision as its second and third arguments.

The editing modes shown in the following chart correspond to the choices provided by the AutoCAD UNITS command. *Precision* is the number of digits to the right of the decimal point that we wish to have displayed.

1	Scientific
2	Decimal
3	Engineering (decimal inches)
4	Architectural (fractional inches)
5	Arbitrary fractional units

If called with only one argument, `rtos` converts it according to the current LUNITS and LUPREC values set by the UNITS command:

```
Command: (rtos 17.5)
"17.5000"
```

`rtos` allows us to override the UNITS settings for the conversion of a given number. Regardless of the UNITS setting, the following examples show various ways in which a number can be converted into a string:

```
Command: (rtos 17.5 1 4)                ;Scientific
"1.7500E+01"

Command: (rtos 17.5 2 2)                ;Decimal
"17.50"
```

```
Command: (rtos 17.5 3 2)                ;Engineering
"1'-5.50""

Command: (rtos 17.5 4)                  ;Architectural
"1'-5 1/2""

Command: (rtos 17.5 5)                  ;Fractional
"17 1/2"
```

The precision field is ignored for modes 4 and 5.

Once a number has been converted to a string it can be handed to the AutoCAD TEXT command and placed in our drawing:

```
Command: (command "text" '(3 3) 0.2 "" (rtos 44.5))     ;44.5 is placed in drawing
nil                                                      ; at 3,3
```

`distof` str [int]	Takes a string that represents a real number and converts it to that number.

`distof` (***DIS**tance **TO F**loat*) performs the reverse conversion of `rtos`. Its optional second argument is an integer that specifies the mode in which the string is formatted. The permissible modes are the same as shown with `rtos`. If omitted, the current value of LUNITS is used; this may be incompatible, however, and cause `nil` to be returned.

Let's use `rtos` to convert a number into a string in both decimal and architectural units:

```
Command: (setq len (rtos 1.889 1))
"1.889"

Command: (setq len2 (rtos 1.889 4))
"1 7/8""
```

Now let's convert these strings back to numbers with `distof`:

```
Command: (distof len)
1.889

Command: (distof len2)
nil

Command: (distof len2 4)
1.875
```

```
Command: (getvar "lunits")
2
```

Because LUNITS is set to 2 (Decimal) we can translate the decimal string without specifying the mode, but we're unable to do so with the string in architectural units. Note also that architectural strings are approximations that may not convert back to the exact number with which we began.

`cvunit` num/list str1 str2	Converts a number or a list of numbers from the unit of measurement specified by str1 to the unit of measurement specified by str2.

```
Command: (cvunit 1 "pound" "grams")
453.592

Command: (cvunit '(1 1.5 2) "day" "hours")          ;Convert entire list
(24.0 36.0 48.0)
```

When we need to perform multiple conversions, it is quicker to convert 1.0 once, then multiply. For example, we can convert three inches to centimeters as follows:

```
Command: (setq base (cvunit 1 "in" "cm"))
2.54

Command: (* base 3)
7.62
```

The accepted unit types are stored in the file `acad.unt`, and we can add our own conversions to that file. A myriad of conversions (and their abbreviations) are supported, but they must make sense. For example, we can't convert feet into grams!

`fix` num	Takes a number and converts it into an integer.
`float` num	Takes a number and converts it into a real number.

These two number conversion functions are explained with the arithmetic functions in Section 3.2.

9.2.3 Angle Conversions

`angtos` ang [int] [int]	Takes an angle (in radians) and converts it into a string. It accepts an optional editing mode and a precision as its second and third arguments.

The editing modes shown in the following chart correspond to the choices provided by the AutoCAD UNITS command. Note, however, that the numbers range from 0 to 4 (consistent with the values accepted by the AUNITS system variable), whereas in UNITS they range from 1 to 5. *Precision* is the number of digits to the right of the decimal point that we wish to have displayed.

0	Degrees
1	Degrees/minutes/seconds
2	Grads
3	Radians
4	Surveyor's units

If called with only one argument, `angtos` converts it according to the current AUNITS and AUPREC values set by the UNITS command:

```
Command: (angtos pi)
"180"
```

`angtos` allows us to override the UNITS settings for the conversion of a given angle. Regardless of the UNITS setting, the following examples show various ways in which an angle can be converted into a string. Note the use of `dtr` in these examples.

```
Command: (angtos (dtr 28.45) 0 4)          ;Degrees
"28.4500"

Command: (angtos (dtr 28.45) 1 3)          ;Degrees/minutes/seconds
"28d27'0""

Command: (angtos pi 2 2)                   ;Grads
"200.00g"

Command: (angtos pi 3 5)                   ;Radians
"3.14159r"

Command: (angtos (dtr 28.45) 4 3)          ;Surveyor's units
"N 61d33'0" E"
```

If a negative angle is supplied, it is converted to a positive angle in the reverse direction:

```
Command: (angtos (dtr -270))
90
```

As with `getangle`, `angtos` is affected by the settings of ANGBASE and ANGDIR. If we set ANGBASE to north, the angle is measured from there:

```
Command: (angtos (dtr 40))
"310"
```

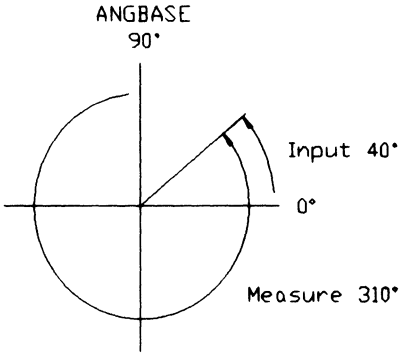


Figure 9-1

If we now set ANGDIR to 1 (clockwise), the same angle is measured in a clockwise direction:

```
Command: (angtos (dtr 40))
"50"
```

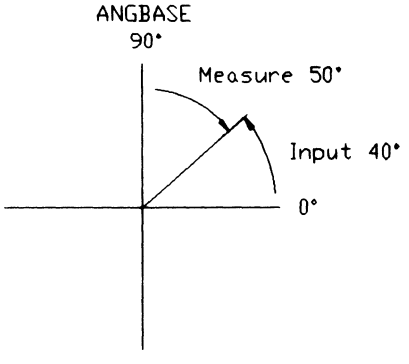


Figure 9-2

Once an angle has been converted to a string, it can be handed to the AutoCAD TEXT command. Let's say that the variable `ang` has been bound to the radian equivalent of 28.45 degrees and that ANGBASE and ANGDIR have been reset to east and counterclockwise, respectively. Then the following function will put 28.45 into our drawing.

```
Command: (command "text" '(3 3) 0.2 "" (angtos ang 0 2))
nil
```

`angtof` str [int]	Takes a string representing an angle and converts it to that angle (in radians).

`angtof` (***ANG****le* ***TO F****loat*) performs the reverse conversion of `angtos`. The optional second argument to `angtof` is an integer that specifies the mode in which the string is formatted, and the permissible modes are the same as shown with `angtos`. If omitted, the current value of AUNITS is used. This may be incompatible and cause `nil` to be returned.

Let's use `angtos` to convert an angle (in radians) into both degrees/minutes/seconds and surveyor's units:

```
Command: (setq deg (angtos (dtr 90.4567) 1 4))
"90d27'24""

Command: (setq surv (angtos (dtr 90.4567) 4))
"N 0d W"
```

Now let's convert these strings back to radians:

```
Command: (angtof deg)
1.57887

Command: (angtof surv)
nil

Command: (angtof surv 4)
1.5708

Command: (getvar "aunits")
0
```

Because AUNITS is set to 0 (Degrees) we can translate the degree string without specifying the mode, but we're unable to do so with the string in surveyor's units. Note also that surveyor's strings are approximations that may not convert back to the exact number with which we began.

9.2.4 Point Conversions

`trans` pt from to {`t`}	Translates the point from the coordinate system indicated by the second argument to the coordinate system indicated by the third argument.

The coordinate systems for the *from* and *to* arguments can be expressed as:

- One of the following integer codes:

  0 WCS
  1 Current UCS
  2 DCS for the current viewport, and current model space viewport when used with code 3
  3 Paper space DCS (used only with 2)

- An entity name.

- A 3D extrusion vector.

In addition to WCS and UCS, `trans` also recognizes Entity (ECS), Display (DCS), and Paper Space Display (PSDCS) Coordinate Systems.

An ECS is a coordinate system relative to an entity we select from our drawing. It is also called the Object Coordinate System (OCS) in Release 13.

The DCS is the coordinate system in which images are displayed on our screens. We can convert a point to DCS when we want to learn how it will appear to the operator.

The PSDCS can be used to translate points from paper space to the DCS of the active model space viewport (only).

We supply the optional fourth argument of `t` to specify that the first argument is a displacement instead of an actual point. 2D points can also be translated.

See Section 10.3 for a practical example using `trans`.

9.3 Measurement Functions

In AutoCAD we frequently measure angles, lengths, and distances. This section introduces subrs that our programs can use to perform such measurements.

`inters` pt pt pt pt [`nil`]	Takes the endpoints of two lines and returns the point where they intersect.

Let's draw a line from point `a` (2 2) to point `b` (2 8), and another line from point `c` (1 4) to point `d` (6 4).

```
Command: (setq a '(2 2) b '(2 8) c '(1 4) d '(6 4))
(6 4)                                          ;Set point values; last one is returned
```

We can use `inters` to determine the intersection point, as follows:

```
Command: (inters a b c d)
(2.0 4.0)
```

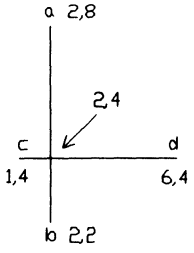


Figure 9-3

The arguments to `inters` are the endpoints of lines, but `inters` doesn't require that we draw these lines.

If all four supplied points are 3D points, then `inters` returns a 3D intersection. However, if *any* point is 2D, then `inters` returns a 2D intersection.

Now let's change our drawing a little by making `c`'s value be the point `5,4`:

```
Command: (setq c '(5 4))                       ;Change c
(5 4)
```

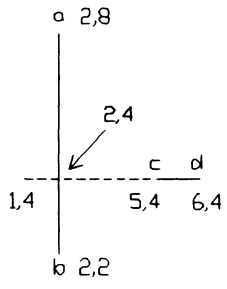


Figure 9-4

Now when we run `inters`, what happens?

```
Command: (inters a b c d)
nil
```

`inters` returns `nil` because the two lines don't intersect.

`inters` takes an optional fifth argument. If it's `nil`, then the lines are considered to be infinite in length and the intersection point is returned, *even if it's beyond the actual endpoint of one or both lines:*

```
Command: (inters a b c d nil)
(2.0 4.0)

Command: (setq c '(5.9 3))                ;Alter c to make lines
(5.9 3)                                   ; nearly parallel

Command: (inters a b c d nil)             ;Lines still meet
(2.0 -36.0)
```

The first call to `inters` extends the line and returns the point `2,4`. The second time we run `inters` it returns a point that is off the screen.

`osnap pt str`	Applies an object snap mode specified in the string to the supplied point and returns the resultant point.

The second argument to `osnap` is a string that specifies one or more object snap modes separated by commas. The function decides which snap mode locates a point nearest to the supplied point and returns that point. To see how `osnap` works, let's draw a line from 2,2 to 2,8:

```
Command: (command "line" '(2 2) '(2 8) "")
nil
```

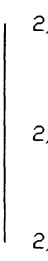


Figure 9-5

We can now use osnap to find the midpoint or endpoint of this line:

```
Command: (osnap '(2 7) "midp,endp")     ;2,7 is nearest to the endpoint
(2.0 8.0 0.0)

Command: (osnap '(2 6) "midp,endp")     ;2,6 is nearest to the midpoint
(2.0 5.0 0.0)
```

osnap considers the supplied point and returns the nearest endpoint or midpoint of any entity that passes through that point. 2,7 is nearer to the line's endpoint, and 2,6 is nearer to its midpoint.

There are several places where we can run into trouble using osnap:

- When specifying these snaps we must say "midp" and "endp"; "mid" and "end" don't work.
- The mode string cannot contain spaces. If we enter

  ```
  Command: (osnap '(2 7) "midp, endp")
  nil
  ```

  osnap doesn't locate a point.
- Suppose we enter

  ```
  Command: (osnap '(2 7) "midp,endp")
  ```

  and the point 2,7 is equidistant from the endpoint of one line and the midpoint of another. We cannot be certain where this will snap. Theoretically, this function could return different results on different days.

To be safe we should limit the mode string to only one snap and be sure that the supplied point lies on only one entity. For example, we might use `osnap` to locate a line's endpoint if the user fails to snap when he selects the line. Suppose the user clicks on a point near a line that runs from 2,2 to 2,8. Then we can find the line's endpoint in the following manner:

```
Command: (setq pt (getpoint "Select end point: "))   ;User clicks near line
(2.0 3.5767 0.0)

Command: (osnap pt "endp")
(2.0 2.0 0.0)
```

When the user clicks near the line, `osnap` accepts the point and snaps to the endpoint of the line. Note, however, that if the crosshairs are beyond the aperture range, `osnap` returns `nil`.

`polar` pt ang num	Takes a point, an angle (in radians), and a distance as arguments. It returns a point at the specified angle and distance from the supplied point.

The `polar` function works in a manner very similar to standard polaring in AutoCAD. The angle is supplied in radians with respect to the current construction plane.

```
Command: (polar '(1 1) (dtr 90) 2)
(1.0 3.0)
```

When we call the `polar` function with the point 1,1, an angle of 90 degrees, and a distance of 2, it returns the point 1,3, which is two units due north of the X axis from 1,1.

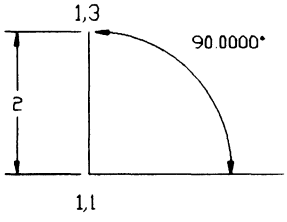


Figure 9-6

`angle` pt pt	Takes two 2D points and returns the angle between them (in radians).

```
Command: (angle '(1.0 1.0) '(1.0 4.0))
1.5708
```

The angle from 1,1 to 1,4 is 90 degrees, which is 1.5708 radians. If 3D points are supplied, they are projected into the current construction plane:

```
Command: (rtd (angle '(5 1 1) '(3 1 4)))
180.0
```

The angle from 5,1 to 3,1 is 180 degrees; the Z coordinate is ignored. We frequently use the `rtd` function with `angle` in order to see the result in degrees.

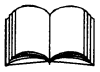
From the examples in this chapter you might get the mistaken impression that we constantly convert between degrees and radians. In practice we typically obtain an angle from the user in degrees and convert it to radians, perform all of our calculations, then convert the result back to degrees.

`distance` pt pt	Takes two points and returns the distance between them.

```
Command: (distance '(1 1 1) '(1 1 2))
1.0
```

The distance between 1,1,1 and 1,1,2 is 1.0. If one or both points is 2D, then a 2D measurement is made. For there to be a 3D measurement, both points must be 3D. A Z value of 0.0 is not assumed:

```
Command: (distance '(1.0 1.0 3.0) '(1.0 1.0))
0.0
```



The `c:long-dist` command in Section 6.3 shows a practical use for distance.

Example

`c:midln` is a useful command that demonstrates four of the measurement functions: `osnap`, `polar`, `angle`, and `distance`. It asks the user to select two lines, then draws a third line midway between the first two. (`c:midln` was written by Todd Moen and enhanced by Keith Thorslund.)

Let's say we've drawn the following two lines:

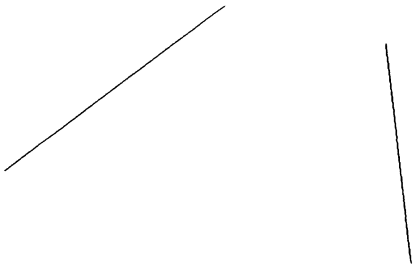


Figure 9-7

We'd run the `c:midln` command as follows:

```
Command: midln
Select First Line: (User clicks on one line)
Select Second Line: (User clicks on the other line)
```

`c:midlin` draws a third line between the first two, as shown in Figure 9-8:

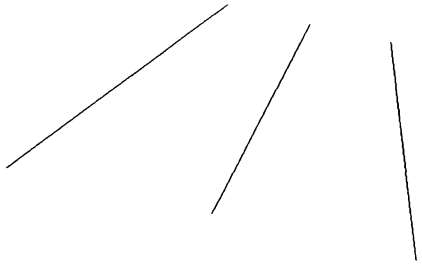


Figure 9-8

The starting point of this new line is midway between the starting points of the other two lines. Likewise, the ending point is midway between the other lines' ending points. The command itself is fairly lengthy due to a couple of error-prevention steps:

1. It sets the value of the APERTURE and PICKBOX system variables to 5. This lessens the possibility that the user will select a point that is off the line, thereby causing the program to crash.

2. It verifies that the nearest endpoints are always measured. If the opposing endpoints are compared, the following line will be drawn instead:

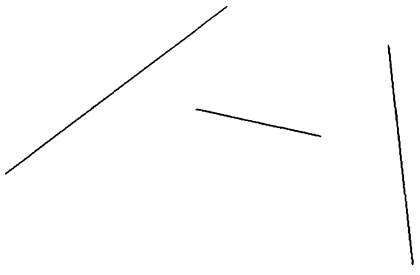


Figure 9-9

Let's examine the code itself:

```
1 (defun c:midln (/ ap pb ln1 ln1e1 ln1e2 ln1md ln2 ln2e1 ln2e2 ln2md temp 1stmp 2ndmp)
    (setq ap (getvar "aperture")
          pb (getvar "pickbox"))
    (setvar "aperture" 5)                          ;Set aperture and pickbox the same
    (setvar "pickbox" 5)

2   (setq ln1 (entsel "\nSelect First Line: ")
          ln2 (entsel "\nSelect Second Line: ")
          ln1e1 (osnap (cadr ln1) "end")           ;ln1 endpt nearest pick point
          ln1md (osnap (cadr ln1) "mid")           ;ln1 midpt
          ln2e1 (osnap (cadr ln2) "end")           ;ln2 endpt nearest pick point
          ln2md (osnap (cadr ln2) "mid"))          ;ln2 midpt

    ;;calculate other endpoint of each line
3   (setq ln1e2 (polar ln1e1 (angle ln1e1 ln1md)(* (distance ln1e1 ln1md) 2.0))
          ln2e2 (polar ln2e1 (angle ln2e1 ln2md)(* (distance ln2e1 ln2md) 2.0)))

    ;;make ln2e1 be the ln2 endpt nearest to the ln1 pick point.
4   (if (>  (* (distance ln1e1 ln2e1)              ;Is product of opposite
             (distance ln1e2 ln2e2))               ; distances greater?
          (* (distance ln1e1 ln2e2)                ;(cross product of vectors)
             (distance ln1e2 ln2e1)))
        (setq temp ln2e1)                          ;yes, switch ln2e1 and ln2e2
              ln2e1 ln2e2
              ln2e2 temp)
    )
5   (setq 1stmp (midpoint ln1e1 ln2e1)             ;Set endpoints of midline
          2ndmp (midpoint ln1e2 ln2e2))
    (command "line" 1stmp 2ndmp "")                ;Draw midline
    (setvar "aperture" ap)                         ;Restore system variables
    (setvar "pickbox" pb)
    (prin1))

  ;;;Function to calculate the midpoint of two supplied points.
  (defun midpoint (pt1 pt2)
    (mapcar '(lambda (a b) (/ (+ a b) 2.0)) pt1 pt2))
```

Figure 9-10 shows the settings for the variables used in `c:midln`:

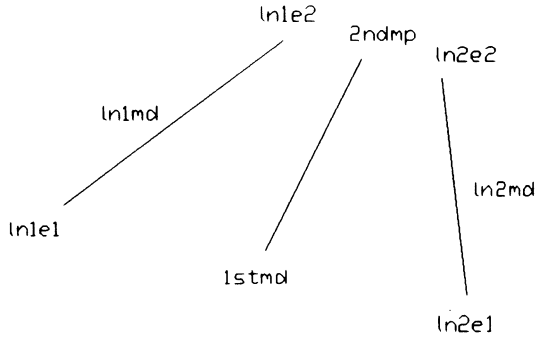


Figure 9-10

1. At the start of `c:midln` the APERTURE and PICKBOX system variables are saved away and reset to 5. This increases the chances that the user will correctly select a line. Note, however, that the routine is unreliable when the two lines are less than five pixels apart.

2. The `entsel` function (defined in Section 10.2) twice asks the user to select a line, and in each case returns a list whose *second* element is the pickpoint. Thus the following four functions must access the *cadr* of the lists that `entsel` returns. These functions use `osnap` to locate both the endpoint nearest to the pickpoint and the midpoint of each line.

3. The two calls to `polar` find the other endpoint of each selected line. For each line the `angle` function determines the angle from the first endpoint to the midpoint, and `distance` returns the distance between the two. The latter is multiplied by 2, which yields the line's total length. Finally, `polar` returns a point at the calculated angle and distance from the line's starting point; this is the other endpoint of the line.

4. This section of code finds a cross product of two vectors. This is where the function verifies that `ln1e1` and `ln2e1` are the two endpoints closest to each other. If they're not, it transposes the values of `ln2e1` and `ln2e2`.

5. Finally, `c:midlin` calls the `midpoint` function to calculate the midpoint between each set of endpoints. It then draws a line between the two midpoints, resets the system variables, and exits gracefully. (`midpoint` is explained in Section 7.3).

Note: Although this function provides an excellent example of the measurement functions, endpoints of lines are usually more easily accessed through

the entity data. Chapter 10, Lab Exercise 6 has us rewrite this command using entity manipulation functions.

9.4 Symbol Manipulation Functions

In addition to `setq`, there are two somewhat esoteric functions that manipulate AutoLISP symbols.

`set` sym obj	Makes the object be the value of the symbol. It returns the object.

From its definition, `set` appears to work exactly like `setq`. The difference is that `setq` is a special operator that inhibits evaluation of its first operand, whereas `set` is a normal function that sends both of its operands to `eval` for evaluation. `set` requires that the first argument returned by `eval` be a symbol.

A couple of examples will demonstrate the difference between these two functions. We'll start with `setq`:

```
Command: (setq y 3)
3

Command: (setq x 'y)
Y

Command: (setq x 5)
5

Command: !x
5

Command: !y
3
```

The first two forms initialize `y` to have a value of `3` and `x` to have a value of `y`, giving us these relationships:

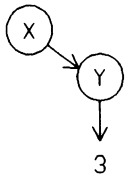


Figure 9-11

When we enter `(setq x 5)`, `x` is given a new value but `y` is left untouched. `x` and `y` end up with the values `5` and `3`, respectively:

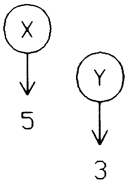


Figure 9-12

Now we'll run the same code, this time substituting `set` for `setq` in the third form:

```
Command: (setq y 3)
3

Command: (setq x 'y)
Y

Command: (set x 5)
5

Command: !x
Y

Command: !y
5
```

In this example, we again initialize `x` and `y` to `y` and `3`, respectively. When we enter `(set x 5)`, the operands `x` and `5` are handed to `eval` Level 2 for evaluation. The arguments `y` and `5` are then given to the `set` function, which binds `y` to `5`; `x` remains unchanged:

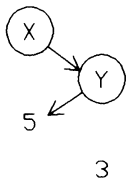


Figure 9-13

The special operator `setq` requires that its first operand be a symbol; that operand is not evaluated. The normal function `set` *does* evaluate its first operand, and the argument that is returned *must* be a symbol.

`set` is a very important subr in certain situations. Suppose we're writing a function that uses the parameter `s`. When the function is run, `s` will be bound to, say, the symbol `length`, which the user will supply as an argument. How can we have our function set the value of `length` to 8.4?

We can't write `(setq length 8.4)` because `length` is an argument; the next time the function is called a different symbol will be supplied. Nor can we write `(setq s 8.4)` because that will change the value of `s`, not `length`. Instead we state `(set s 8.4)`. This binds the *value of s,* that is, `length`, to 8.4, which is what we want.

If we quote the first operand to `set`, then it works identically to `setq`:

```
Command: (setq z 2)
2

Command: (set 'z 2)
2
```

In fact, the word `setq` is merely an abbreviation for ***SET Q***uote.

There is one further difference between `setq` and `set`: Whereas one `setq` can assign values to an unlimited number of variables, `set` can only perform a single variable assignment.

Example

One task we frequently perform is to set the values of system variables within the commands that we write. For example, we might turn off command echoing to make our command run more cleanly. If we wish to program in an elegant manner, we should save the current value of CMDECHO when we enter the function and restore it when we exit.

The `setv` and `rsetv` functions (written by Tony Hotchkiss) automate this process for us. We use `setv` to give a system global variable a new value and `rsetv` to restore its previous value. Suppose we're writing a command that draws various entities and we want to turn off CMDECHO so that prompts aren't printed on the screen. We can add the following two lines to our command:

```
(defun c:draw-obj ()
  (setv "cmdecho" 0)
        .
        .
        .
  (rsetv "cmdecho"))
```

Assume that the initial value of CMDECHO is 1. Let's run `setv` at the Command: prompt and see how it works:

```
Command: (getvar "cmdecho")
1

Command: (setv "cmdecho" 0)
0

Command: (getvar "cmdecho")
0
```

`setv` changes the value of the supplied system global variable. It also saves its previous value so that `rsetv` can restore it later. In the code's comments, below, CMDECHO is used as an example, but `setv` works with any system global variable whose value we are permitted to change.

```
1 (defun setv (sysvar newval / cmdnam)
2   (setq cmdnam (read (strcat sysvar "1")))          ;Create CMDECHO1
3   (set cmdnam (getvar sysvar))                      ;Save CMDECHO's value there
4   (setvar sysvar newval))                           ;Then set CMDECHO to new value
```

1. When we call `setv`, `sysvar` is bound to the string `"cmdecho"` and `newval` is bound to the number 0.

2. `strcat` joins the strings `"cmdecho"` and `"1"` to form `"cmdecho1"`, then `read` converts it to the symbol `cmdecho1`. `setq` assigns this new symbol to `cmdnam`:

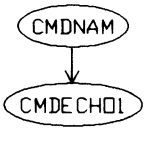


Figure 9-14

3. `getvar` obtains the current value of `cmdecho`, 1, and `set` assigns it to `cmdecho1`:

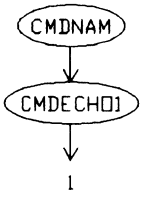


Figure 9-15

If setq had been used instead of set, 1 would have been assigned to the local variable cmdnam. Since cmdnam needs to be evaluated, set must be used instead of setq.

4. The final line simply sets CMDECHO to the supplied value, 0. The original value has been saved in the new global variable, cmdecho1, where it can be retrieved by the rsetv function.

Here's how we'd run rsetv at the Command: prompt:

```
Command: (rsetv "cmdecho")
1

Command: (getvar "cmdecho")
1
```

Let's examine this function:

```
  (defun rsetv (sysvar / cmdnam)
1   (setq cmdnam (read (strcat sysvar "1")))       ;Create CMDECHO1
2   (setvar sysvar (eval cmdnam)))                 ;Restore CMDECHO's value
```

1. The first line of rsetv is identical to the first line of setv; it builds the symbol cmdecho1 and assigns it to cmdnam, as shown in Figure 9-14.

2. The interesting code is on the second line. In the innermost parentheses the operand, cmdnam, is evaluated, and its value, cmdecho1, is handed to the eval function. eval evaluates cmdecho1 and returns its value, 1. Thereafter setvar assigns 1 to the value of sysvar, cmdecho.

Notes

- We call setv and rsetv within AutoCAD commands that we write, rather than at the Command: prompt. Before they are called, however, they must already be defined. Thus, they are excellent candidates for the acad.lsp file.

- set is frequently used in conjunction with read and/or eval. Situations arise where we need to create symbols and give them values. Our *modus operandi* is to use getstring to obtain a symbol name from the user, then have read convert the supplied string to a symbol. To set the symbol's value we must use set because we don't actually have the symbol in our hand; we have a parameter whose value is that symbol.

Later, when we need to learn or change the value of that symbol, we again have to go through the parameter. We explicitly call `eval` with the parameter as the operand. The double evaluation provides access to the value of the new symbol rather than the symbol itself.

A few pages hence we'll see another practical example using `set`. But first, get settled in your seat, because now *we're gonna pay a call on the* `atoms-family`*!*

Da-da-da-dum dum dum, da-da-da-dum dum dum. . . . Enter the glorious mansion known as AutoCAD, wind your way past the gooey dialog boxes, and descend the stairway to the swirling depths of AutoLISP. There you will find some gleaming newer functions that have replaced the cobweb-covered features of releases past.

Gomez opens a vault and, amid some functions that are most mysterious and spooky, proudly points to his proof that someone at Autodesk possesses a wonderful sense of humor. With a twinkle in his eye he presents his prized function, the `atoms-family`!

`atoms-family` is a function that returns a list of the built-in AutoLISP symbols, as well as any other symbols that we have created during the current drawing session.

Gomez describes the chaos that formerly reigned at the mansion: " . . . symbols all over the place. I could never get hold of them. The worst part was when Thing would walk down `atomlist` searching for a certain symbol. We'd always have to run the `c:clean` command to get rid of the mess.* Now that they've done away with `atomlist` . . ."

"*Done away with* `atomlist`*?!!*" you ask in a shocked tone. "Certainly," replies Gomez, with a look of polite superiority crossing his face, "Autodesk always said that its demise was only a matter of time. It was replaced in Release 12 by the `atoms-family` function. Let me explain how the function works."

* Prior to Release 12 `atomlist` was bound to a list of all symbols in AutoLISP. Veteran AutoLISP programmers are familiar with the `c:clean` command, which was used to purge `atomlist` of unneeded symbols and free up valuable memory. In the process it would close all opened files.

`atoms-family` 0/1 {list}	Returns a list of symbols present in AutoLISP. The first argument specifies whether the returned value is a list of symbols or a list of strings.

When called without the optional list, `atoms-family` returns a list containing the names of all of the bound symbols. These include the following:

- All of the built-in functions, that is, the subrs
- Symbols (including function and command names) that are defined in the `acad.lsp` file and loaded when we initiate a drawing session
- Symbols that we've created during the current drawing session

If a symbol is unbound, `atoms-family` returns `nil` in its place. In other words, it lists all of the symbols that used to be on `atomlist` except for those whose values are `nil`. There's a lot more of them than there used to be on `atomlist`, however, due to the introduction of ADS and dialog box functions.

If the first argument is 0, `atoms-family` returns a list of symbols:

```
Command: (atoms-family 0)
(X Y Z ...)
```

If, on the other hand, the first argument is 1, then a list of strings is returned instead:

```
Command: (atoms-family 1)
("X" "Y" "Z" ...)
```

Rather than having `atoms-family` return all of the currently defined symbols, we can specify particular symbols we want it to find. The optional second argument is a *list of strings* that name the symbols we want returned. As before, these symbols are returned as symbols or strings, depending on the setting of the initial integer argument.

For example, suppose we assign values to the variables `x` and `y`, but not `z`. Then we can call `atoms-family` in the following ways:

```
Command: (setq x 1 y 2 z nil)
nil
```

```
Command: (atoms-family 0 '("x" "y" "z" "car"))
(X Y nil CAR)

Command: (atoms-family 1 '("x" "y" "z" "car"))
("X" "Y" nil "CAR")
```

Morticia is upset that we no longer have access to `atomlist`. "How are we ever going to clean up all those nasty little symbols that hang around the attic and antagonize the bats?" she asks. "We used to be able to wipe out all those symbols we no longer needed."

"Oh, don't worry," replies Gomez. "We are at an exalted station in life where memory is cheap. We aren't about to fill it up like we did in the old Release 2.5 days. And once we have the symbols, we can easily determine `atoms-family` *values.*"

"Oh Gomez, you're always plugging your movies!"

"Uuuuuuuurrrrrrrrrgh," says Lurch.

"You're right about that, my friend. The `c:clean` command has gone the way of my dearly departed great-great-great-great-great-grandfather."

"But Gomez, darling, I used `c:clean` to close the files I accidentally left open. If Pugsly ever got into them, I don't know what I'd do!"

"Don't worry, my dear, the invaluable `atoms-family` function enables us to take care of that situation too. Just type in the following command."

```
(defun c:close-files ()
  (foreach elt (atoms-family 0)            ;Sequence through all symbols
    (if (equal (type (eval elt)) 'file)    ;Is symbol's value of type FILE?
        (set elt (close (eval elt)))))     ;Yes, close it.
  (prin1))
```

"It's certainly shorter than the old `c:clean` command. But can you explain the code to me . . . like you used to in the old days?"

Gomez inhales deeply, filling his senses with the scent of Morticia's perfume. An eager look appears in his eyes.

"You see, my one and only," he begins, "the `foreach` code accesses each symbol that the `atoms-family` function returns. Since we didn't supply the optional list of symbols, `atoms-family` returns a list of every bound symbol in our drawing."

"But won't that take forever to run?"

"Not at all. To test it I created over a thousand symbols, and it still only took a second to search the entire list and close all of the opened files." Gomez continues: "The `if` statement then checks whether the symbol names a file and, if so, it closes that file."

"You're going too fast," interrupts Morticia, rather breathlessly now. "I need for you to go slow. Now tell me, what's the purpose of those `evals`? And the `set` . . . didn't you mean to use `setq`?"

"Well, this is a bit tricky," admits Gomez, "although when I explained it to Uncle Fester, his lightbulb did go on! But it's difficult for me to continue with these people looking on."

Not wishing to bother our good friends, I suggest that we leave them alone in the vault and repair to the wine cellar, break open a few bottles, and examine the `c:close-files` command in a bit more detail.

Gomez certainly got off to a good start by explaining that the `foreach` allows us to sequentially access every defined symbol. Each time through this loop the next symbol becomes the value of the local variable `elt`.

Suppose that the value of `elt` is the symbol `ff`, and that `ff` names a file that needs to be closed. In other words, the value of `ff` is a file descriptor. These relationships can be shown pictorally as follows:

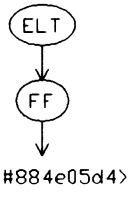


Figure 9-16

In AutoLISP we can test whether a symbol's value is a file descriptor with the `type` function. But if we run the `type` function on the value of the symbol `elt`, it returns `sym`, because the value of `elt` is always a symbol . . . in this case the symbol `ff`:

```
Command: (type elt)
SYM
```

What we really want to do is run `type` on the *value of the value* of `elt`; in other words, we need to supply a level of indirection. To accomplish this we must hand `elt` explicitly to the `eval` function:

```
Command: (eval elt)
<File: #884e05d4>

Command: (type (eval elt))
FILE
```

Note that we must say `(eval elt)` on the next line as well, for exactly the same reason.

Now we can compare the result of the `type` function with the symbol `file`. If there isn't a match, then no action is performed and `foreach` goes on to the next symbol. If they do match, we know that the particular symbol names a file and the test succeeds. The `if` takes the THEN path, and the next line is evaluated.

`(eval elt)` returns the symbol `ff`, to the `close` function, which closes the file and returns `nil`. Recall from Section 8.3, however, that if we run `close` without a `setq`, the value of `ff` remains the file descriptor rather than becoming `nil`. This is what we call a *dead file descriptor:* It can no longer access the file. Running code that uses this file descriptor causes a "file not open" error.

The `c:close-files` command is a case in point. If the user had previously closed a file without `setq`ing the symbol to `nil`, `if`'s test succeeds. The code then attempts to close the nonopen file, causing an error. For simplicity, this command assumes that the file is open.

To prevent the `c:close-files` command itself from causing this problem, we must set `ff`'s value to `nil` when we close the file. However, if we use `setq` instead of `set`, then the value of the symbol `elt` becomes `nil`, like so:

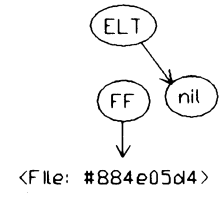
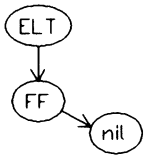


Figure 9-17

This is not our desired result. The value of `elt` is the symbol `ff`, and we want `ff`'s value to become `nil`. In other words, we want the *value of the value* of `elt` to become `nil`. Again, we need that level of indirection.

This brings us back to the `set` function. What this line of code actually says is, "Take the value of the symbol `elt` (i.e., `ff`) and make its value be `nil`." Thus `ff` gets set to `nil`, as shown in Figure 9-18:



<File: #884e05d4>

Figure 9-18

At this moment, Gomez and Morticia emerge from the vault. From the broad smile of satisfaction on their faces, we must conclude that Gomez has successfully imparted the seeds of his knowledge to his wife.

"On the last line of the `c:close-files` command, the function `(prin1)` with no arguments returns nothing," concludes Gomez, with a triumphant wave of his hand. "It simply allows a function to exit gracefully, as we ourselves are about to do."

"Oh darling, you're wonderful! Once again I'm reminded of why I married you."

"It's easy, my sweet one, with the crazy and the kookie `atoms-family`!"

"Uuuuuuuurrrrrrrrrrgh".

Da-da-da-dum dum dum, da-da-da-dum dum dum. . . .

9.5 Graphics Functions

The following AutoLISP functions enable our programs to modify the graphics screen.

`redraw`	Redraws the current viewport and returns `nil`.

In its simplest form, `redraw` works exactly like the AutoCAD REDRAW command.

We can use `redraw` to highlight, blank out, or redraw individual entities. The full contract of `redraw` is discussed in Section 10.3.

`graphscr`	Switches to the graphics screen on a single-screen system and returns `nil`.

Running `graphscr` is equivalent to hitting the `F1` key in the DOS version of AutoCAD, or `F2` in the Windows version.

`textscr`	Switches to the text screen on a single-screen system and returns `nil`.

Running `textscr` is equivalent to hitting the `F1` key in the DOS version of AutoCAD, or `F2` in the Windows version.

`textpage`	Switches to the text screen on a single-screen system, clears the screen, and returns `nil`.

`textpage` is the same as `textscr` except that it clears off the screen. We can even eliminate the `nil` that's returned by entering the following code:

```
Command: (progn (textpage) (princ))
nil
```

Since the Windows version of AutoCAD displays our command history, `textpage` does not clear off the text screen. In Windows, `textpage` works exactly like `textscr`.

`vports`	Returns a list of lists that describe the current viewport configuration. Each inner list consists of the ID number and the screen position of one viewport.

In AutoLISP we can split our graphics screen into multiple viewports. We can also find out information about the viewports and make a specific viewport current.

We create viewports in AutoLISP by supplying the AutoCAD VIEWPORTS (or VPORTS) command as an argument to the `command` function. For example, we can divide our graphics screen into two horizontal viewports as follows:

```
Command: (command "vports" 2 "h")
nil
```

Every viewport has an identification number, which is a unique integer assigned to the viewport by AutoCAD. We can learn the ID number of the current viewport by examining the system variable, CVPORT:

```
Command: (getvar "cvport")
1
```

Similarly, we can use `setvar` to alter the value of this variable, thereby changing the current viewport.

How do we learn the ID numbers of all active viewports? In AutoCAD we can specify the `?` option to the VPORTS command. In AutoLISP we use the `vports` function:

```
Command: (vports)
((1 (0.0 0.5) (1.0 1.0))
 (3 (0.0 0.0) (1.0 0.5)))
```

The list returned by `vports` contains the configuration information for our two horizontal viewports. The coordinates of viewports are always between 0 and 1, where the lower left corner of the screen's graphics area is at 0,0 and its upper right corner is at 1,1.

The `vports` function always lists the current viewport first. The order of the other viewports in the list is totally arbitrary. In our example, the current viewport's lower left corner is at 0.0, 0.5 and its upper right corner is at 1.0, 1.0. Its ID numbers is 1.

Note that although the vports function has the same name as the AutoCAD VPORTS command, the two have very different contracts: The VPORTS command creates viewports, whereas the vports function returns the current viewport configuration.

vports comes in handy when it's not practical to have the operator click with the pointing device to select a new current viewport. A slide show in a script file is an instance where we wouldn't want operator intervention.

The vports function can be used with the CVPORT variable to change the current viewport:

```
Command: (setq ports (vports))
((1 (0.0 0.5) (1.0 1.0)) (3 (0.0 0.0) (1.0 0.5)))

Command: (setq a (caar ports))
1

Command: (setq b (caadr ports))
3

Command: (setvar "cvport" b)
3
```

The vports function returns the list of active viewports, which is assigned to the variable ports. Next, the symbols a and b are bound to the ID numbers of each viewport. Finally, one of the ID numbers is assigned to the CVPORT system variable, making that viewport current.

Note: We can't set the CVPORT variable to a nonactive viewport, that is, one that's in a saved viewport configuration.

9.6 Advanced Topic: Display Control and Device Access

AutoLISP provides a group of functions that enable us to control our display and the devices that interface with it. With these functions we can

- draw lines that don't become entities;
- read and write to the menu areas;
- determine cursor coordinates, tablet selections, or nonprinting characters entered by the user; and
- read and modify digitizer calibrations.

These are tasks that most AutoLISP programmers may never need to perform. If you are just starting to learn AutoLISP, I suggest coming back to this section once you've gained some expertise in the language. The functions will make more sense if you type in the examples as you read this section.

`grclear`	Clears the current viewport and returns `nil`.

`grclear` flips to the graphics screen and blanks out the current viewport but doesn't erase any entities or touch the menu and status areas of the screen. We might use it to momentarily conceal our drawing and overlay it with a message or temporary drawing.

```
Command: (grclear)
nil
```

The original entities' images are restored when we REDRAW or REGEN.

`grdraw` pt pt int {int}	Draws a vector between two points in the specified color and returns `nil`.

`grdraw` is used to draw a vector in the current viewport. The vector looks like a line but it's temporary; a line entity is *not* added to the entity database. Thus we can't supply this vector to MOVE or any other AutoCAD command that issues a Select objects: prompt. Furthermore, if we leave the graphics window or perform a zoom, the vector disappears.

The points may be 2D or 3D. If we supply a point that is off the screen, AutoCAD "clips" the vector to fit the screen.

The third argument is an integer between 0 and 255 that corresponds to AutoCAD's standard color codes. It can also be –1, in which case the vector is drawn in *XOR ink*. XOR ink draws over other objects in a complementary color and erases itself when overdrawn.

A few examples will show how `grdraw` works:

```
Command: (command "color" 1)                    ;Make the default color be red
nil

Command: (command "line" '(2 2) '(8 2) "")      ;Draw a red line
nil
```

```
Command: (grdraw '(2 3) '(8 3) 1)                    ;Draw a red vector
nil

Command: (grdraw '(3 2) '(7 2) -1)                   ;Overdraw a portion of the red line
nil                                                  ; in XOR ink

Command: (command "color" 2)                         ;Change default color to yellow
nil

Command: (command "line" '(4 2) '(5 2) "")           ;Overdraw vector with a
nil                                                  ; yellow line

Command: redraw                                      ;Goodbye vectors!
nil                                                  ;Only the red and yellow lines remain
```

The optional fourth argument to `grdraw` is an integer that, if positive, indicates that the vector is to be highlighted. Normally a highlighted vector is drawn as a dashed line, but this depends on the display device itself. The following function draws a highlighted green vector from 5,1 to 3,8:

```
Command: (grdraw '(5 1) '(3 8) 3 1)
nil
```

`grvecs` list {list}	Draws a series of vectors between points in the supplied list and returns `nil`.

`grvecs` is used when we want to draw multiple vectors. Its first argument is a *vector list,* which we provide in the following format:

({color} from to {color} from to . . .)

Each entry consists of an optional color code followed by the starting and ending points of the vector. Color codes range from 0 to 255 and correspond to AutoCAD's standard color codes. If a color value is greater than 255, then the vector is drawn in XOR ink, which draws over other objects in a complementary color and erases itself when overdrawn. If the color code is negative, the vector is highlighted. If the color code is absent, the vector is drawn in the default or previously listed color.

The points may be 2D or 3D. If we supply a point that is off the screen, AutoCAD "clips" the vector to fit the screen. In the following example, indentations are for readability only:

```
Command: (command "line" '(4.5 6) '(6 6) "")          ;Draw line in default color
nil

Command: (grvecs '(  (3 6) (3 3)                      ;Default color
                   1 (3 3) (4 2)                      ;Red
                     (4 2) (8 5)                      ;Also red
                  -1 (8 5) (6 6)                      ;Still red but now highlighted
                 300 (6 6) (3 6)))                    ;XOR ink

nil
```

In this call, `grvecs` draws a vector between 3,6 and 3,3 in the current color. It then draws red vectors from 3,3 to 4,2, from 4,2 to 8,5, and (highlighted) from 8,5 to 6,6. The final vector is drawn in XOR ink, so that the right half, which overlays the line, is drawn in a complementary color, and the left half is drawn in black. Had we supplied a point that was off the screen, the vector would have ended at the screen's boundary.

`grvecs` takes an optional *transformation matrix* as its second argument. This is a list that `grvecs` uses to change the location and/or proportion of the vectors. The list has four elements, each of which is itself a list of four real numbers. The identity transformation matrix looks like this:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The first three rows indicate transformations, such as scaling and rotation, for the X, Y, and Z coordinates of each vector point. The last row must be present but is currently unused. The last column is used to specify a translation to be applied to each point. The ways in which we use the transformation matrix are most easily understood by looking at some examples.

Example 1

```
(defun c:grvecs1 ()
  (grvecs '(1 (1 1) (3 3))
          '((1 0 0 2)                ;X translation 2
            (0 1 0 3)                ;Y translation 3
            (0 0 1 0)                ;Z
            (0 0 0 1))))
```

The first argument tells `grvecs` to draw a red vector from 1,1 to 3,3. The transformation matrix specifies an X translation of 2 and a Y translation of 3. Thus each vector point is offset by 2,3 and, as a result, the vector is actually drawn from 3,4 to 5,6.

Example 2

```
(defun c:grvecs2 ()
  (grvecs '(1 (1 1) (3 3))
          '((4 0 0 0)                ;X scaled 4
            (0 1 0 0)                ;Y
            (0 0 1 0)                ;Z
            (0 0 0 1))))
```

In this example we supply the same vector points as before, but there is no translation. Instead, an X scaling of 4 is specified. This causes the X coordinate of each vector point to be multiplied by 4, yielding a red vector from 4,1 to 12,3.

Example 3

```
(defun c:grvecs3 ()
  (grvecs '(5 (1 1 0) (3 3 1)
            6 (2 7 0) (1 2 1))
          '((4 0 0 0)                ;X scaled 4
            (0 3 0 -2)               ;Y scaled 3 and offset -2
            (0 0 1 0)                ;Z
            (0 0 0 1))))
```

`c:grvecs3` draws blue (5) and magenta (6) vectors between 3D points. Each X coordinate is multiplied by 4 and each Y coordinate is multiplied by 3. Following the multiplication, each Y coordinate is translated by –2.

In this example the vectors are actually drawn from 4,1,0 to 12,7,1 and from 8,19,0 to 4,4,1. Since the point 8,19,0 is beyond the drawing limits, the vector is clipped at the edge of the screen (or viewport).

`grtext` int str {int}	Takes an integer code representing the status line or a screen menu area and writes a string in that area. If successful, it returns the string; otherwise it returns `nil`.

`grtext` is most commonly used to write a string into one of the boxes in the screen menu area. The total number of boxes available on a particular screen is stored in the SCREENBOXES system variable.

```
Command: (getvar "screenboxes")      ;We can write to boxes in
24                                   ; the range 0-23
```

The first argument to `grtext` ranges from 0 to one less than this value, where 0 indicates the topmost screen menu box. For example, we can write the string `"Angle=40"` in screen menu box 7, which is the eighth box in the screen menu area:

```
Command: (grtext 7 "Angle=40")       ;Put the string "Angle=40" in
"Angle=40"                           ; box number 7
```

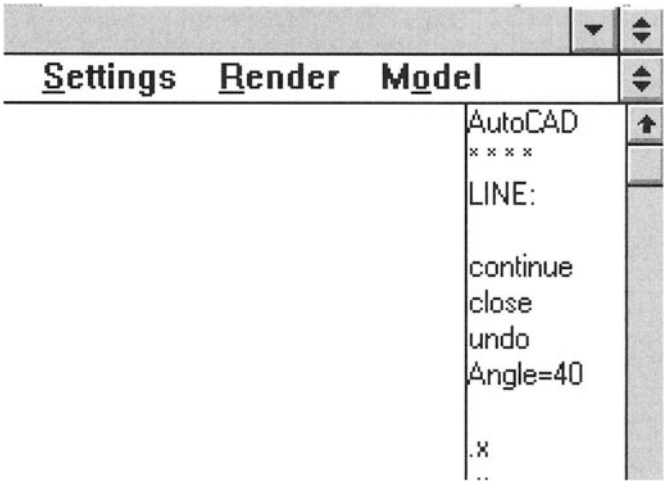


Figure 9-19

grtext takes a third optional integer argument. If it's a positive number, the specified entry is highlighted. There's a trick to using this argument on many platforms: If we want to write a new entry, we may have to enter and highlight it in two separate steps:

```
Command: (grtext 0 "EFC" 1)          ;Highlights entry 0 but
"EFC"                                ;  may NOT put "EFC" in the box

Command: (grtext 0 "EFC")            ;If it doesn't, then first put "EFC"
"EFC"                                ;  in the box,

Command: (grtext 0 "" 1)             ;  then highlight it
""
```

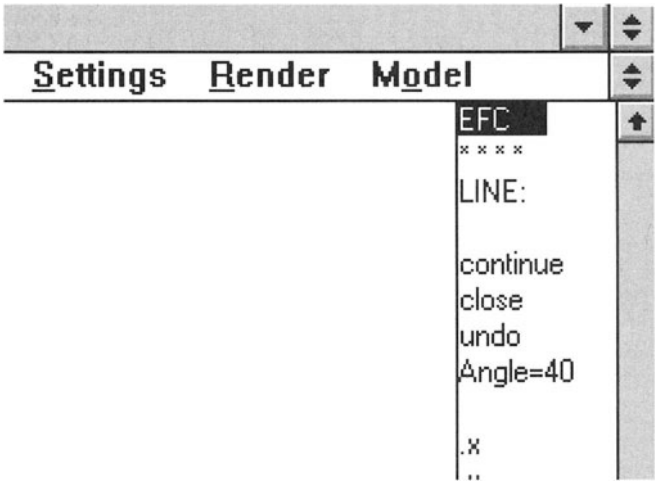


Figure 9-20

Only one box can be highlighted at any given time. If we highlight a box, any box that is already highlighted will be dehighlighted. If we supply a highlight argument of zero, any highlighted entry is dehighlighted.

The first argument to grtext can be –1 or –2, in which case the string is written to the mode status line area or coordinate status line area, respectively. These options ignore the third argument, if present:

```
Command: (grtext -1 "This is my mode status area")
"This is my mode status area"
```

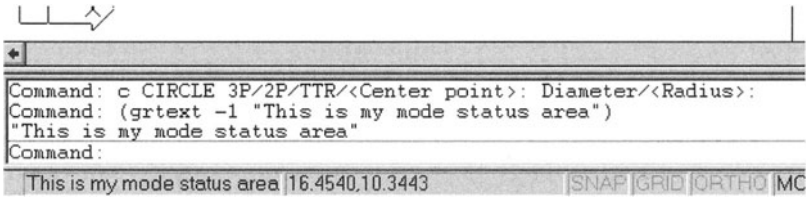


Figure 9-21

If coordinate tracking is enabled, the text supplied with a –2 entry will be instantly overwritten by the coordinates. We must turn off coordinate tracking to have this value remain.

Notes

- For all boxes, if the supplied string is too large, it is truncated; if it's too small, it is padded to the right with spaces.
- If we call `grtext` with no arguments, all boxes are restored to their original settings.

Once we've written text into the screen menu area with `grtext`, we can detect the user's selection of that text with `grread`.

`grread {t/nil}` {int} {int}	Reads values from an input device and returns a list that indicates the device and the value that it received.

`grread` is an intricate input function that can read information from the keyboard or pointing device (i.e., mouse or digitizer cursor). It tells us which keyboard key the user pressed or where on the drawing screen, menu, or digitizer tablet he clicked.

`grread` returns a two-element list. The first element is an integer code that indicates the type of device that was used for input. The second element is either an integer or a list, depending on the input device. The following table summarizes the values that `grread` might return:

Input Code		*Second Element*	
Value	*Device/Input*	*Value*	*Description*
2	Keyboard	Integer	Character code
3	Selected point	3D point	Point coordinates
4	Screen/Pull-down menus (Selected with screen pointing device)	0 to 999	Screen menu box number
		1001–1999	POP1 menu box number
		2001–2999	POP2 menu box number
		...	...
		16001–16999	POP16 menu box number
5	Pointing device	3D point	Drag mode coordinates
6	BUTTONS menu	0–999	BUTTONS1 menu button number
		1000–1999	BUTTONS2 menu button number
		2000–2999	BUTTONS3 menu button number
		3000–3999	BUTTONS4 menu button number
7	TABLET1 menu	0–32767	Digitized box number
8	TABLET2 menu	0–32767	Digitized box number
9	TABLET3 menu	0–32767	Digitized box number
10	TABLET4 menu	0–32767	Digitized box number
11	AUXILIARY menu	0–999	AUX1 menu button number
		1000–1999	AUX2 menu button number
		2000–2999	AUX3 menu button number
		3000–3999	AUX4 menu button number
12	Pointer button	3D point	Point coordinates

Here are a few examples of how we might run the function:

```
Command: (grread)                    ;User types "a"
(2 97)

Command: (grread)                    ;User clicks on drawing screen
(3 (2.453 6.8792 0.0))

Command: (grread)                    ;Screen menu box #2
(4 2)

Command: (grread)                    ;Pulldown menu 5, box #11
(4 5011)

Command: (grread)                    ;Tablet area 1, box #15
(7 15)
```

grread takes three optional arguments. If the first argument, called the *track*, is t, then grread tracks the pointing device. It returns a list whose car is 5 and whose cadr is the coordinates of the current pointing device position:

```
Command: (grread t)                    ;Current crosshairs position
(5 (5.55243 7.9343 0.0))
```

The second optional argument to grread is a bit value that modifies the function's behavior. These bits are analogous to the initget bits for the get functions and, like the initget bits, may be added together to effect multiple modifications. The grread bit values are shown in the following table:

1	Return cursor's location. This is identical to setting the track argument to t.
2	Return all keyboard values, including cursor keys. This option enables grread to read such keys as Up-arrow and F3. It also prevents the cursor from moving when the user presses a key.
4	Modify the cursor according to the value of the third optional argument, which can be one of the following: 0 Display normal crosshairs 1 Don't display any cursor 2 Display an entity selection "target" box
8	Don't display the "console break" message when the user presses <esc>.

Here's how we might use the grread bits:

```
Command: (grread nil 1)                ;Current crosshairs position
(5 (4.3233 6.98282 0.0))               ;Same as (grread t)

Command: (grread nil 2)                ;F3 key pressed
(2 27)

Command: (grread nil 4 1)              ;No crosshairs or cursor displayed
(3 (4.233 11.0013 0.0))                ;  But user can still click

Command: (grread nil 8)                ;User presses <esc>
error:                                 ;"console break" msg not displayed,
(GRREAD nil 8)                         ;  but function still aborts
```

The grread bits can be added together to effect more than one constraint. The following example puts no prompt on the screen (bit value 4, option 1) and reads cursor keys (bit value 2):

```
Command: (grread nil 6 1)          ;Down-arrow key pressed
(2 -5)
```

Notes

- When using the grread bits, the track argument *must* be set to nil. Otherwise the track overrides the grread bits and returns the current pointing device position:

```
Command: (grread t 2)          ;1st argument overrides second, so
(5 (8.8663 7.233 0.0))         ;  pointing device coordinates returned

Command: (grread 2)            ;1st argument is non-nil,
(5 (1.9838 5.239 0.0))         ;  which is the same as t
```

- The *AutoLISP Reference* states that the track argument returns coordinates when the pointing device is moved. In DOS, and with some mouse drivers in Windows, grread returns the pointing device's coordinates at the time the function is executed, regardless of whether the device is actually moved.

  On some machines, if the cursor is already positioned in the pulldown menu area, grread waits until the user moves the pointing device onto the drawing area, then returns the first screen coordinates it encounters. If instead of moving the mouse onto the drawing screen the user selects a popup menu choice, grread ignores the track argument.

- Although a grread bit value of 8 suppresses the "console break" message for <esc>, the function still aborts and returns the "error:" part of the message.

- Documentation prior to Release 13 lists a bit value of 16. This bit value is not supported.

- grread occasionally returns a list whose car is 13. This has no meaning and should be ignored.

Here's an example that uses grread to read a password entered by the user. It was written by Keith Thorslund.

```
1 (defun c:pass (/ st let)
    (setq st "")
    (prompt "\nEnter access code: ")
2   (while (/= (setq let (cadr (grread nil 2))) 13)        ;Until user hits enter
      (princ "*")                                          ;  print * and
      (setq st (strcat st (chr let))))                     ;  add letter to string
3   (if (not (member st #pwdlist))                         ;Is string in password list?
        (princ "Invalid access code. "))                   ;No, print error message
    (princ))
```

Prior to running the c:pass command, we must assign a list of password strings to the global variable #pwdlist, as follows:

```
Command: (setq #pwdlist (list "Jeffrey" "Graham" "Lynda" "Taffy"))
("Jeffrey" "Graham" "Lynda" "Taffy")
```

When we run the command, it appears as follows:

```
Command: pass
Enter access code: *****
nil
```

Let's examine the code:

1. The local variable st will be bound to the string the user enters. It is initialized to the null string, then the user is prompted to input a password.

2. grread is used for input instead of getstring because we want to read the string character by character. read-char is also inappropriate because it requires a terminating (<space> or <enter>) following each input character.

   The second argument to grread, 2, allows cursor keys such as F3 to be entered. Passwords having nonprinting characters are harder to steal.

   When the user presses a key, its ASCII character code is assigned to the variable let. The while loop is repeated until the user hits <enter> (ASCII code 13).

   Each time through the loop a "*" is printed instead of the character in order to keep it hidden. grread is the only input function that provides such a mechanism for secrecy. The character code is converted into a string with the chr function, then appended to the string st with strcat.

3. When the string has been completely built (the user pressed <enter>), `c:pass` checks whether it is a member of the password list. If not, an error message is printed.

If you'd like to use this command in a program, you might want to make a couple of modifications:

- This is an ideal situation in which to use the `exit` or `quit` subr; it would replace `princ` at the end of the command.
- Perhaps add an outer loop to give the user two or three attempts before causing the function to crash.

♦ ♦ ♦

Two more functions work with our menus and digitizer tablet. `menucmd` sets and retrieves menu items and status. `tablet` is used to set and read digitizer calibrations.

`menucmd str`	Sets or retrieves a menu item's contents or status according to the command string argument. It returns either a menu item status or `nil`.

`menucmd` allows us to control the menus as we would in AutoCAD. The allowable command strings are the same as we'd put in a menu macro, except that the leading $ is omitted. The following menu sections are supported:

String values	*Menu section*
B1–B4	BUTTONS menu 1 through 4
A1–A4	AUX menus 1 through 4
P0–P16	Pulldown menus 0 through 16
I	Image tile (icon) menu
S	Screen menu
T1–T4	TABLET menus 1 through 4
M	DIESEL string expressions

We can use menucmd to specify where we'd like a menu placed in one of the following ways:

```
Command: (menucmd "s=osnap")          ;Display OSNAP menu
nil                                   ;  in screen menu area

Command: (menucmd "p4=sub")           ;Put submenu "sub" into
nil                                   ;  pulldown menu 4
```

Once a submenu has been assigned to a menu section we can have it displayed on the screen as follows:

```
Command: (menucmd "p4=*")             ;Display pulldown menu 4
nil
```

As we do in menu files, we can use AutoLISP to check and alter the status of menu entries. Here are a few examples:

```
Command: (menucmd "p4.1=~")       ;Disable (gray out) pulldown
nil                               ; menu 4 label 1

Command: (menucmd "p4.1=?")       ;Return current status of this label
"~"                               ;Item is disabled

Command: (menucmd "p4.1=#?")      ;Return item number and status
"P4.1=~"                          ;Item is disabled

Command: (menucmd "p4.1=!.")      ;Place check mark in front of item
nil

Command: (menucmd "p4.1=?")       ;Return status of this item
"!"                               ;Label is checked, but no longer
                                  ;  grayed out. Label can't be both.
```

tablet 0/1 {pt} {pt} {pt} {pt}	Reads the current digitizer calibrations or uses the supplied points to calibrate the tablet.

If the first argument to tablet is 0, it returns the current digitizer calibrations. If it is unable to do so (e.g., the digitizer isn't a tablet or it hasn't been calibrated), tablet returns nil.

If the first argument to tablet is 1, then the second must be four 3D points. The first three points specify the three rows of the tablet's transformation ma-

trix. The fourth point is a direction vector. These arguments are explained in detail in the *AutoLISP Reference.*

The simplest way to use the `tablet` function is to first call the AutoCAD TABLET command to calibrate the digitizer. Next, invoke `tablet` with an argument of 0 to obtain those calibrations, and save them in a file. Later, when those settings need to be restored, simply hand the saved list to the `tablet` function.

9.7 Background Topic: Application-Handling Functions

Because they're interpreted rather than compiled, AutoLISP programs run more slowly than programs written in other computer languages. This speed difference, however, is not noticeable for most of the functions and commands that we write. Furthermore, the AutoLISP interpreter provides interactive development and debugging capabilities unavailable in other high-level languages. Thus, AutoLISP is the appropriate tool for writing small routines that automate daily tasks, gather information, and modify entities.

Third-party developers often write their code in C and other programming languages. These programmers create complex or number-crunching applications, and they need the speed that only compiled programs can offer.

The *AutoCAD Development System* (ADS) is a facility that allows us to write and compile programs in other high-level languages, then load them into AutoLISP. Three AutoLISP subrs support ADS:

`xload` string [obj]	Takes the name of a compiled ADS application, loads the program into memory, and returns the name of the application.

`xload` is analogous to `load` and, like `load`, evaluates the optional second argument if the `xload` fails.

`xunload` string [obj]	Takes the name of an ADS application, unloads it from memory, and returns the name of the application.

`xunload` evaluates the optional second argument if the unload fails.

`ads`	Returns a list containing the names of all loaded ADS applications.

The application name (or the complete path) is returned as a string.

♦ ♦ ♦

`startapp` str [str]	Takes strings representing a Windows application and a filename and runs the application on the file. It returns a positive integer if it succeeds, 0 if it fails.

`startapp` is a Release 13 function.

`startapp` is an ADS application that is only available in AutoCAD for Windows because it starts a Windows application. The *AutoLISP Reference* gives a good example of its use:

```
Command: (startapp "notepad" "acad.lsp")
33
```

This starts the Windows Notepad application and opens the `acad.lsp` file. We can open a variety of different Windows applications, such as Solitaire:

```
Command: (startapp "sol")
33
```

Please don't tell your boss where you saw this one!

9.8 Labs

1. Write a function that accepts a string and returns a copy of the string with the characters in reverse order.

```
Command: (str-rev "abcde")
"edcba"
```

2. Write a function that takes a string and returns the same string minus the leading spaces.

```
Command: (dss "          abc")
"abc"
```

   Next, write a function to delete the trailing spaces.

```
Command: (dse "abc          ")
"abc"
```

   Finally, write a function that deletes all leading and trailing spaces.

```
Command: (ds "    abc       ")
"abc"
```

3. Alter the `c:sparse` command (shown in Section 9.1) to determine whether an extracted part of the string represents a number. If it does, make the list element be the number rather than the string. Only perform this transformation on numbers.

```
Command: sparse2
Input string: ab 33 cd 44 ef 55
("abc" 33 "cd" 44 "ef" 55)
```

4. Write a command to print a date and time stamp in a drawing. The system global variable CDATE contains the current date expressed as a real number. Convert this variable to a string, format it nicely, and print it at the point specified by the user.

```
Command: stamp
Start point:
```

   When the user enters a point, `c:stamp` puts a string such as `"4/28/95 3:14:51"` at that point.

5. Write a function that checks whether a string is a valid U.S. zip code. Zip codes can be either five numeric characters or five numeric characters, a hyphen, and four numeric characters.

```
Command: (zip "02139")
T

Command: (zip "43210-4438")
T

Command: (zip "123456")
nil
```

6. The `write-char` function comes in handy when we need to write data to a file character by character. But the argument to `write-char` must be the ASCII equivalent of a character. Write a function that takes a string and returns a list of the ASCII values for each of the characters in that string. Remember that spaces are valid string characters.

```
Command: (stoa "a b c")
(97 32 98 32 99)
```

7. Write a command that prompts the user for a starting number or letter. Then, in a loop, have it prompt for a point. On each pass through the loop it should print the next succeeding number or letter at the specified point.

```
Command: nl-seq
Enter starting number or letter: j
Enter point: 3,3 (command puts "j" at 3,3)
Enter point: 5,2 (command puts "k" at 5,2)
Enter point: 1,6 (command puts "l" at 1,6)
```

   Had the user initially entered 10, the command would have inserted 10, 11, and 12 into the drawing.

8. A colleague who works in the mapping field needs a function that takes two numbers expressed as strings, such as `"50"` and `"23"`. He wants the function to prompt the user for a point and put a table in his drawing at that point. The table should show the entered coordinates, hyphenated, as well as the sectors surrounding those coordinates, like so:

```
Command: (surround "50" "23")
Select starting point for table:
```

When user enters a point, the following table is drawn at that point:

49–24	50–24	51–24
49–23	50–23	51–23
49–22	50–22	51–22

9. The I/O chapter introduced the `\t` control character, which tabs over a set amount of spaces. This is inadequate when we need to list items in columns because if one item has too many characters, the columns are thrown off.

Write a function that takes an AutoLISP object and a count of the maximum number of spaces in a column. Determine the number of characters in the object (the `otos` function we wrote in Chapter 8, Lab Exercise 5 is useful here) and subtract it from the count. The remainder is the number of spaces that the function should return. For example, if the object is the number 234 and the count is 10, the function should return 7 spaces.

Since it's very hard to count spaces on the screen, have the function initially return a printing character. Once it's debugged, replace that character with a space. For example, if the function prints an `"x"` instead of a space, it runs as follows:

```
Command: (tab 234 10)
"xxxxxxx"

Command: (tab 'abcd 6)
"xx"
```

10. Write a function that takes a list of AutoLISP objects and a numeric tab setting and prints the elements in columns. Have it call the `tab` function that we wrote in Lab Exercise 9. In the following example the tab character is `"x"` instead of `" "` to show the spacing.

```
Command: (columns '(red green yellow blue) 8)
REDxxxxxGREENxxxYELLOWxxBLUExxxx
```

11. Write a command that lists all the bound symbols and what kind of data they are bound to.

```
Command: syms
Integers: (A)
Real numbers: (B PI)
Symbols: (T)
Strings: (PAUSE C #F#)
Point Lists: (MID2 MID1 LN1E LN2E LN2S LN1S)
File Descriptors: nil
Entity Names: (E)
Selection Sets: (SS1 SS2)
```

Notes

- You will need to use `atoms-family` in this exercise.
- User-defined functions are implemented as lists. Our solution tests whether the first element of a list is a number, in which case it assumes that the list is a point.
- We haven't yet seen entity names and selection sets; these categories can be added later.
- If there are no symbols in a particular category, print `nil`.

12. Write a command to search the viewport configuration. Make the viewport whose lower left corner is at 0,0 be the current viewport.

IV

Programming AutoCAD's Entity Database

One of the main advantages of AutoLISP is its ability to access the database of entities we have drawn. *This capability is not available via the standard AutoCAD commands,* and it's where the true power of AutoLISP lies. Most AutoLISP users find that fully half of the programs they write include one or more entity-handling functions.

Via the entity functions we can learn such information as the starting and ending points of lines, the layer on which an entity resides, and the entities contained in a block.

In Chapter 10 we'll learn how to access the entity database to rapidly change such entity data as widths of polylines and radii of circles, how to conveniently explode and reblock inserts, and many other useful, time-saving tasks.

AutoLISP enables us to group entities into selection sets and make changes to all of them with a single command. This facility is even more powerful than Release 13 groups because we can make changes to the entity database itself.

In Chapter 11 we'll see programs that quickly change the color of selected entities, alter block attributes, scale selected inserts, create text styles on the fly, and temporarily thaw layers. We'll discover how functions that manipulate selection sets can save an enormous amount of time compared with doing these tasks in AutoCAD.

Entities are actually easier to learn and use than much of what we have studied so far. But to use them effectively we must have a good understanding of the concepts and information presented to this point.

10

Accessing and Modifying Entities

Topics Covered in This Chapter

- Introduction to the AutoCAD entity database
- Entity name functions and entity data functions
- Entity handles
- Advanced topic: extended entity data

Goals for This Chapter

- Understand what entities are and how they are represented in the AutoCAD entity database.
- Learn how to create and access entities by their names.
- Learn how to write AutoLISP routines that manipulate entity data.
- Learn how to extract information about entities from the entity database.
- Learn how to program changes to entities and their properties.

Introduction

AutoLisp gives us tremendous control over the geometry in our drawing by allowing us to access and change entity data. There are two kinds of entities: graphical and nongraphical. A *graphical entity* is an object that we draw on the screen and includes the following familiar objects (among others):

Lines	Circles	Points	Text
Polylines	Arcs	Block Inserts	Attributes
Dimensions	Solids	3D Faces	Mtext (Rel. 13)

AutoLisp provides a way to access these entities from within our programs, to read and even change information about them. For example, let's say we have a plate with 300 holes in it, and suddenly some of those holes need to be ¼" bigger in radius. In AutoCAD we would have to find and change these holes one at a time. In AutoLISP we can say "Find all of the red holes on Layer 2," or "Find all of the holes with a radius of 1.4 in this crossing window and increase the radius by .25."

In AutoLISP we can group entities into selection sets to manipulate several entities as a single unit, much as we do with a COPY or MOVE command. We can also access nongraphical entities such as Block and Layer tables. Selection sets and tables are occasionally mentioned in this chapter and are covered in detail in the next chapter.

10.1 Understanding the Entity Database

Nearly everything that is drawn on the screen is a reflection of AutoCAD's entity database. To effectively use the entity subrs, we must first understand the composition of the entities themselves. Suppose we draw a line from 2,3 to 4,5:

```
Command: line
From point: 2,3
To point: 4,5
To point:
```

This line is an entity and, like all entities, has an entity name. An *entity name* is an AutoLISP object type that we haven't seen before. It has the following printed representation:

```
<Entity name: 88450510>
```

The numbers in the printed representations of entity names are hexadecimal and are assigned in the order that entities are created. As with functions and file descriptors, entity names can be printed by AutoLISP, but we can't type them in. For this reason we never use an entity name directly. Instead, we `setq` a variable to the entity name and access the entity name via that variable.

Entity names can be used for a variety of tasks. When we run an AutoCAD command that asks us to select objects, such as ROTATE or ERASE, instead of clicking on an entity we can simply supply its entity name.

An entity name is also used to access the *entity data,* which is an association list consisting of the name of the entity and other descriptive information. For example, the entity data for the line we drew might appear as follows:

```
((-1. <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (100 . "AcDbLine") (10 2.0 3.0 0.0) (11 4.0 5.0 0.0) (210 0.0 0.0 1.0))
```

The car of each data entry is a *group code* (which is akin to a DXF code) that tells us what the data in the cdr of the pair means. Some group codes are common across all entity types. Others are particular to specific entities. Still others are present in several entity types but have different meanings in each.

Commonly used entity codes are listed in Appendix E. A complete group code listing for each entity type can be found in the *AutoLISP Reference.* Release 12: Appendix B; Release 13: Chapter 16.

The following group codes are present in every graphical entity in the order shown:

–1	The entity name of the entity.
0	A string that specifies the entity's type.
5	A string that specifies the entity's handle.
100	A string that specifies the entity's subclass data marker.
67	0 if the entity is in model space; 1 if it's in paper space.
8	A string that specifies the layer on which the entity resides.

As of this writing, group code 100 is unused. In Release 12, group code 100 doesn't exist and group codes 5 and 67 are optional.

Thereafter, the entities diverge. A given line will have pretty much the same group codes as any other line, and circles will be quite similar to one another. But lines and circles will differ radically.

For the line we drew, the group codes have the following meaning:

–1	Entity name
0	This entity is of type LINE
5	Entity's handle is 22
100	Subclass data marker
67	Line is in model space
8	Line is on Layer 0
10	Line's starting point is 2, 3, 0
11	Line's ending point is 4, 5, 0
210	Extrusion direction vector

10 and 11 fields always indicate points. In this case they are the starting and ending points of the line.

The 210 field denotes the line's extrusion direction. If an entity has a thickness, this field denotes the direction that the entity extrudes from the X-Y plane. The default is 0,0,1, meaning that unless the entity is in a UCS with a particular extrusion direction, it extrudes straight out from the X-Y axis (thus we won't see its thickness).

Notice that there are no entries for the linetype or the color of the line. Why is that?

The linetype and color information is set BYLAYER and is kept in the Layer table. If we change the color of a line to differ from BYLAYER, then there *will* be a dotted pair for the line's color. Making this information BYLAYER greatly reduces the amount of data AutoCAD must store for each entity.

Whenever we create a new entity on the screen, the entity information is recorded in the AutoCAD database. We can then use the entity name and data functions to access and alter the entities in this database. Many of the tasks we're about to see are either cumbersome or impossible to perform in AutoCAD alone but are readily accomplished with the versatile and powerful entity functions. Probably half of the AutoLISP routines we write contain one or more of these functions.

10.1.1 Complex Entities

Some entities are so complex that we can't specify all of the information about them in one entity datum. For example, consider the following polyline:

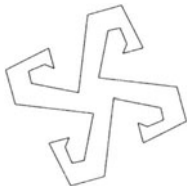


Figure 10-1

If this were a collection of individual lines, we would have a separate entity datum for each line. Each data entry would record the starting and ending point of that line.

A polyline is a single entity comprising multiple connected line segments. If the polyline is closed, then the starting point and the ending point are the same; otherwise they are not. Regardless, it isn't sufficient to record the starting and ending points of the polyline, because they don't provide any information about the various line segments that compose the polyline. To completely specify the polyline data, AutoCAD must keep track of the starting and ending points of each line segment.

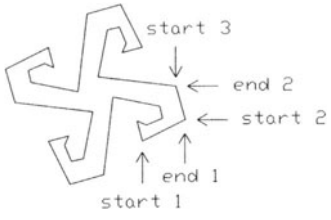


Figure 10-2

Since polyline segments are continuous, each line segment starts at the exact point where the previous line segment ends. Thus, it's redundant to keep track of both the starting and ending points of each line segment. For this reason, the entity data records only the point where two line segments meet. This meeting point is known as a *vertex*.

We can't draw a vertex on its own. A vertex is specifically a *subentity* of a polyline, and there is a separate entity datum for each vertex point in a polyline. Thus, a polyline is said to be a *complex entity*.

Another example of a complex entity is a block insert with attributes attached. Block inserts can have an unlimited number of attributes. Therefore, an attribute is a subentity type as well.

Note: While we're on the topic of blocks let's clarify the difference between a block definition and a block insert. A *block* is a nongraphical entity definition that we create with the BLOCK command; it's never drawn on the screen. An *insert* is a graphical instance of a block that we bring into being with the INSERT command.

When we want a block insert to have an attribute, we create an *attribute definition* with the ATTDEF command, then make that definition part of the block. When the block insert is created, the *attribute* itself is attached to the insert as a subentity. Thereafter we can change, say, the color of an attribute by modifying the subentity. We can also modify the attribute definition itself in the Block table.

♦ ♦ ♦

Let's draw a closed polyline from 1,1 to 3,3 to 4,2 and see how complex entity data are stored in the database:

```
Command: (command "pline" '(1 1) '(3 3) '(4 2) "c")
nil
```

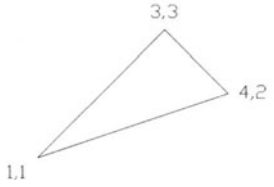


Figure 10-3

Five entity names and data are created, as follows (the ellipses indicate that there are more dotted pairs than are actually shown. Only the relevant data are displayed in order to keep the examples as straightforward as possible):

```
((-1 . <Entity name: 88450510>) (0 . "POLYLINE") (5 . "22") (100 . "AcDbEntity")
 (67 . 0) (8 . "0") (66 . 1) (70 . 1) . . .)

((-1 . <Entity name: 88450518>) (0 . "VERTEX") (5 . "23") (100 . "AcDbEntity")
 (67 . 0) (8 . "0") (10 1.0 1.0 0.0) . . .)

((-1 . <Entity name: 88450520>) (0 . "VERTEX") (5 . "24") (100 . "AcDbEntity")
 (67 . 0) (8 . "0") (10 3.0 3.0 0.0) . . .)

((-1 . <Entity name: 88450528>) (0 . "VERTEX") (5 . "25") (100 . "AcDbEntity")
 (67 . 0) (8 . "0") (10 4.0 2.0 0.0) . . .)

((-1 . <Entity name: 88450530>) (0 . "SEQEND") (5 . "26") (100 . "AcDbEntity")
 (67 . 0) (8 . "0") (-2 . <Entity name: 88450510>))
```

The first entity datum is the polyline *header record.* This is the main entity datum that identifies the entity as a polyline.

Group code 66 has a cdr of 1, which indicates that this is a complex entity and that subentities follow. All polylines and inserts with attributes have a 66 field set to 1. Other entities theoretically have a 66 field set to 0, but it's never present in the data.

Group code 70 is set to 1 to indicate that the polyline is closed. Were the polyline open, then the 70 field would be 0. This saves AutoCAD having to create an association list for the last point, which is identical to the first point.

Subsequent to the polyline header record there is one record for every vertex of the polyline. Each has a 10 group code whose cdr is the vertex point. There's also additional information, such as the polyline's starting and ending widths (40 and 41 fields, respectively), that isn't shown.

The last entry for every complex entity is of type *seqend,* which stands for ***SEQ****uence* ***END.*** This record, shown in its entirety above, has a pair with a group code of –2 whose cdr is the entity name of the header record for the complex entity. If we find ourselves in the middle of a collection of vertices, we can sequence through them until we encounter the seqend, then use the –2 field to locate polyline header record itself.

Some entity functions treat complex entities the same as they would a line, a circle, or any other entity. Other functions treat complex entities as a collection of records. As we study the various entity functions we must be aware of how each function regards complex entities.

10.2 Entity Name Functions

Several functions can be used to retrieve the entity name of an individual entity. Once we have the entity name, we can supply it to any AutoCAD command that asks the user to select an entity. It can also be used to access the entity data, as we shall see in the following section.

You'll understand these functions better if you type them in as we go along. To start, open a new drawing and create the following two lines and a circle:

```
Command: (command "line" '(3 5) '(6 6) '(8 4) ""
                  "circle" '(6 3) 1)
nil
```

entnext [ename]	Returns the name of the first entity in the database. If given an entity name, it returns the name of the next entity in the database.

When called with no arguments, entnext returns the name of the first non-deleted entity in the database.

```
Command: (setq e1 (entnext))          ;First line created
<Entity name: 88450510>
```

We use entnext within a setq to assign the entity name to a variable. By doing so we only need to call entnext without an argument once. Thereafter, if we supply entnext with an entity name as an argument, it returns the name of the *next* entity in the database. In this way we can "walk through" all of the entities in our database:

```
Command: (setq e2 (entnext e1))       ;Second line
<Entity name: 88450518>

Command: (setq e3 (entnext e2))       ;Circle
<Entity name: 88450520>
```

When its argument is the entity name of the last entity in the database, entnext returns nil:

```
Command: (entnext e3)                 ;No more entities
nil
```

Once we have an entity name, we can supply it to any AutoCAD command that asks us to select an object:

```
Command: move
Select objects: !e3                   ;Circle is now selected
1 found
```

If we have a drawing with 500 entities, it would be quite painstaking to have to entnext through them all to get to the 500th. When we need to access the very latest entity that we created, we use the entlast function.

entlast	Returns the name of the last nondeleted main entity in the database.

```
Command: (setq e3 (entlast))          ;Circle
<Entity name: 88450520>
```

If we supply the entity name of the circle to entnext, it returns nil, which tells us that we've reached the end of the database:

```
Command: (entnext e3)
nil
```

Now let's draw the closed polyline from 1,1 to 3,3 to 4,2 that we examined earlier in this chapter.

```
Command: (command "pline" '(1 1) '(3 3) '(4 2) "c")
nil
```

When we run entlast we get a new entity name because there is a new last entity in the database:

```
Command: (setq e4 (entlast))          ;Polyline
<Entity name: 88450528>
```

What happens when we hand that entity name to entnext?

```
Command: (setq e5 (entnext e4))       ;First subentity
<Entity name: 88450530>
```

Surprise! It doesn't return nil as we might expect. entlast always returns the name of the last *main* entity in the database, be it a line, a circle, or a polyline. Usually (entnext (entlast)) returns nil, but if the last main entity is a complex entity, entnext returns the entity name of the first subentity. In this example it returns the entity name of the first vertex of the polyline.

Entities don't have to be on-screen or on a thawed layer to be selected. However, entlast only moves in a forward direction; we can't give it an entity name in order to access the penultimate entity. As we've seen before in AutoLISP, we can't backtrace arrows in a database.

entsel [str]	Prompts the user to select an entity and returns a list consisting of the main entity name and the pick point.

Whereas entnext can only sequence through the entities in the order of their creation and entlast can only access the most recently created entity, entsel allows the user to select any entity on the screen. Its versatility makes it the most frequently used function for learning an entity's name.

entsel can be run in one of two ways. We can simply issue entsel with no arguments, as follows:

```
Command: (entsel)
Select object: (click on the circle)
(<Entity name: 88450520> (5.05867 3.25 0.0))
```

When the user selects the entity, a list is returned containing the entity name and the point that was selected (in UCS coordinates). You must understand and remember a very important fact, so we'll say it big:

entsel RETURNS A *LIST!!*

New users frequently forget this fact and expect entsel to return an entity name, à la entnext or entlast, which leads to a "bad argument type" error. About 90 percent of the time all we care about is the entity name, so we frequently use entsel within a car function:

```
Command: (car (entsel))
Select object: (click on the circle)
<Entity name: 88450520>
```

Occasionally we do need the pickpoint, such as when we want to break, trim, or extend an entity. In this situation we would also supply the optional prompt to override entsel's default "Select object:" prompt:

```
Command: (setq e (entsel "Select a line to break:"))
Select a line to break: mid of (click on first line)
(<Entity name: 88450510> (4.5 5.5 0.0))
```

Because we specified the midpoint object snap, entsel returns the line's midpoint, *regardless of where we actually clicked on the line.* It ignores running osnaps, however. Keep in mind that we still need to use the car and cadr functions to extract the information from the returned list.

♦ ♦ ♦

Like the get functions, entsel accepts keywords established by a previous call to initget:

```
Command: (initget "e1 e2 e3")
nil
```

```
Command: (entsel)
Select object: e3
"e3"
```

The user who has already input values for `e1`, `e2`, and `e3` shouldn't have to click again when she needs to reuse one of them. But if she enters the string `"e3"`, we must use the `read` function to convert it into a symbol, then explicitly call `eval` to get the value of that symbol. The following function will do this for us:

```
(defun getename (/ en)
  (initget "e1 e2 e3")
  (setq en (entsel "Select entity (e1 e2 e3): "))
  (if (eq (type en) 'str)                          ;Did user enter string bound to ename?
      (eval (read en))                             ;Yes, convert to symbol and get value
      (car en)))                                   ;No, get ename from entsel list
```

When we run the function, the user has the option of clicking on an entity or typing one of the specified symbols:

```
Command: (getename)
Select entity (e1 e2 e3): e3
<Entity name: 88450520>
```

♦ ♦ ♦

Let's use `entsel` to select the line segments on the polyline we drew:

```
Command: (car (entsel))
Select object: (click on line segment from 1,1 to 3,3)
<Entity name: 88450528>

Command: (car (entsel))
Select object: (click on line segment from 3,3 to 4,2)
<Entity name: 88450528>
```

No matter which line segment we choose, `entsel` always returns the entity name of the main polyline header record. Occasionally we want to access one of the polyline vertices. When this need arises, we use the `nentsel` function:

```
Command: (car (nentsel))
Select object: (click on line segment from 1,1 to 3,3)
<Entity name: 88450530>

Command: (car (nentsel))
Select object: (click on line segment from 3,3 to 4,2)
<Entity name: 88450538>
```

`nentsel [str]`	Prompts the user to select an entity and returns a list consisting of the entity name (main or subentity) and the pickpoint.

`nentsel` is a *nested* `entsel`. It works similarly to `entsel`, except that it allows selection of nested entities. In each call above it returns a list containing the entity name of the vertex *preceding* the selected line segment. We also use `nentsel` when we want to access an attribute of a block. When `nentsel` is used to select a main entity, it works exactly the same as `entsel`.

`nentsel` can also access an entity that is part of a block insert, even if that insert is nested within another block. The code to accomplish this is quite intricate, however, so it is discussed as an Advanced Topic in Section 10.6.

As with `entsel`, `nentsel` accepts keywords established by a previous call to `initget`.

♦ ♦ ♦

Sometimes we know a point and want to access the entity that passes through that point. When that need arises we can use `nentselp`.

`nentselp [str] [pt]`	Prompts the user to select an entity and returns a list consisting of the entity name (main or subentity) and the pickpoint. If the optional point argument is supplied, `nentselp` returns a list as though the user had selected the point.

`nentselp` is similar to `nentsel`, but it takes an optional point argument. If the point argument is present, `nentselp` doesn't prompt for user input even if a prompt is also supplied. It simply returns a result as though the user entered that point in response to the prompt:

```
Command: (nentselp '(2 2))
(<Entity name: 88450530> (2.0 2.0 0.0))
```

In this example `nentselp` returns a list whose car is the entity name of the polyline vertex preceding the supplied point and whose cadr is the point itself.

If the specified point is in an entity that is nested in a block insert, `nentselp` returns a transformation matrix for point conversions. This matrix is discussed in Section 10.6.

As with `entsel`, `nentselp` accepts keywords established by a previous call to `initget`. However, keywords are ignored if the point argument is supplied.

10.3 Entity Data Functions

Until now the entity functions we've seen work with entity names. The real excitement comes when we use these entity names to access and alter entity data.

`entdel ename`	Takes an entity name and deletes the entity from the drawing database. It returns the entity name.

```
Command: (entdel e3)                    ;Delete the circle
<Entity name: 88450520>
```

If the entity has already been deleted in this drawing session, `entdel` undeletes it.

```
Command: (entdel e3)                    ;Restore the circle
<Entity name: 88450520>
```

An `entdel`eted object is removed from the entity list; `entnext` won't find it. When we undelete `e3`, it's restored to its former place in the entity list. However, if the symbol `e3` has been given a new value, we can't access the entity name, and therefore can no longer undelete the entity.

We can't delete subentities with `entdel`. Use the ATTEDIT or PEDIT command to accomplish this.

We can have fun with `entdel`. Let's create a donut:

```
Command: donut
Inside diameter <0.5000>:
Outside diameter <1.0000>:
Center of doughnut: (click, then press <enter>, to draw 1 donut)
```

Unless your display driver can't support it, the following code will cause the donut to repeatedly flash off and on:

```
Command: (setq e (entlast))
<Entity name: 88450548>

Command: (repeat 200 (entdel e))
<Entity name: 88450548>
```

Pretty exciting, eh? Actually, this technique can be quite useful. Suppose we have a script that demonstrates the workings of a machine and we want to draw the reader's attention to a particular component of that machine. If we flash the entity off and on a bunch of times, it'll attract the reader's attention pretty quickly. Be certain, however, to repeat the loop an *even* number of times!

`redraw` [ename] [int]	Redraws the current viewport and returns `nil`. If called with an optional entity name, it redraws that entity only, according to the mode indicated by the optional integer argument.

As we have seen, `redraw` redraws the current viewport, just like the AutoCAD REDRAW command. But if called with an entity name, it redraws that specific entity only. Unlike `entdel`, `redraw` does *not* undelete or unERASE entities.

`redraw` takes one of the following mode numbers as a second argument:

1	Redraw entity (default)	3	Highlight entity
2	Undraw entity	4	Unhighlight entity

To see how redraw works, enter the following forms:

```
Command: (redraw e1 3)              ;Highlight first line
nil

Command: (redraw e2 2)              ;Undraw second line
nil                                 ;It's still there but we can't see it

Command: (redraw e1 4)              ;Unhighlight first line
nil

Command: (redraw e2)                ;Redraw second line
nil                                 ;1 is the default second argument
```

`entget` ename {list}	Takes an entity name and returns the entity data.

`entget` is probably the most frequently used entity data function. As its name suggests we use it to get the entity data, which is an association list. We can obtain the entity data for our first line as follows:

```
Command: (setq ed1 (entget e1))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 3.0 5.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

The entity data for this line contains the ten pairs that we discussed earlier. How does this compare with the data for the circle?

```
Command: (setq ed3 (entget e3))
((-1 . <Entity name: 88450520>) (0 . "CIRCLE") (5 . "24") (100. "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbCircle") (10 6.0 3.0 0.0) (40 . 1.0) (210 0.0 0.0 1.0))
```

After the first seven dotted pairs, we again find a group code of 10. For a line, this represents the starting point; for a circle, it's the center point. What do you suppose the 40 group code indicates? Correct, it's the circle's radius. This is the minimum information needed to define a circle.

Following the circle, we created a polyline. Use `entnext` to sequence through the pline's entity names and `entget` to examine the pline's data. Compare what's returned with the polyline data we studied in Section 10.1.1.

Once we have the entity data in our hand we can use the `assoc` function to extract a particular dotted pair:

```
Command: (assoc 8 ed1)
(8 . "0")
```

If we only want the layer name rather than the entire dotted pair, we use `assoc` within the `cdr` function:

```
Command: (cdr (assoc 8 ed1))
"0"
```

Occasionally an entity data entry is a list rather than a dotted pair. This in no way affects how we access that entry. We still use `assoc` to access the entire entry and `cdr` to get the value:

```
Command: (assoc 10 ed1)
(10 3.0 5.0 0.0)
```

```
Command: (setq start (cdr (assoc 10 ed1)))
(3.0 5.0 0.0)
```

Don't forget that we must run these functions inside of a `setq` if we want to save the information that's returned.

We can access the data for an entity we just created by entering

```
Command: (entget (entlast))
```

`entlast` returns the name of the last entity, and `entget` gets the entity data.

`entget` takes an optional list argument that is used with extended entity data (see Section 10.5 for more information).

Let's look at a few practical examples that use the functions we've seen so far. If you'd like a challenge, try writing them before studying the code.

Example 1

`c:find-circles` locates all of the circles in the current drawing, then prints the coordinates of the center point and the radius of each. To verify that it captures them all, draw circles on several different layers. Then run `c:find-circles`, as follows:

```
Command: find-circles

        Center point              Radius

    (9.3209 5.79506 0.0)       0.961436
    (5.16674 4.36187 0.0)      0.958669
    (3.89453 7.47749 0.0)      1.54286
```

```
1 (defun c:find-circles (/ en ed obj)
    (terpri)
    (princ "    Center point                  Radius")
    (terpri)
    (setq en (entnext))
2   (while en                                          ;Not nil
      (setq ed (entget en))
3     (if (eq (cdr (assoc 0 ed)) "CIRCLE")             ;Type = circle?
          (progn
            (setq obj (print (cdr (assoc 10 ed))))     ;Center
            (princ (tab obj 34))                       ;Tab over
            (prin1 (cdr (assoc 40 ed)))))              ;Radius
      (setq en (entnext en)))                          ;Get next until nil
    (prin1))
```

1. The `c:find-circles` command begins by printing a heading preceded and followed by a blank line. We could have used `\n` to insert a newline; `terpri` is an equally valid option. It then calls `entnext` with no arguments to set the variable `en` to the first entity name in the database. This is a typical preloop initialization step.

2. The `while` loop exemplifies standard AutoLISP programming style. The test to `while` is to simply evaluate the symbol `en`; the loop is repeated while the value of `en` is not `nil`.

   Each time through the loop, `en` gets the next entity name in the database as its value. If we look ahead to the next-to-last line, we see where `en` is updated to the next entity name. `(entnext en)` returns `nil` when the end of the entity database is reached; `en` is set to `nil` at that point and the `while` loop exits. Until then the loop is repeated for each entity name, and `ed` is bound to the corresponding entity data.

3. The `if` statement also embodies classic AutoLISP coding in that it contains several examples of a programming cliché that we encountered earlier: running `assoc` within `cdr`. `(cdr (assoc 0 ed))` returns the value of the 0 field, which indicates the entity type. The `if` tests whether the entity is of type `"CIRCLE"`. Note that the string must be uppercase or there won't be a match.

   If the test fails, the entity isn't a circle, so the `c:find-circles` command isn't interested in it. It gets the next entity and repeats the loop.

   If the entity is a circle, the command must print the center point (10 field) and the radius (40 field). `print` returns the center point after it's printed, and it becomes the value of the symbol `obj`. `c:find-circles` then calls the `tab` function that we wrote as a lab in Chapter 9. It creates a string containing the proper number of spaces to tab over and hands that string to `princ` for printing.

As we discover more entity-handling subrs we'll find even easier ways to collect this data. However, this example is a nice initial examination of the entity database.

Example 2

The `c:sl` command is quite useful. It prompts the user to select an entity, then makes that entity's layer become the current layer.

```
(defun c:sl (/ layer)
  (setq layer (cdr (assoc 8 (entget (car (entsel))))))
  (command "layer" "s" layer ""))
```

The `c:sl` command is only two lines long, but the first is nested many levels deep. However, one of the main reasons that Autodesk developers chose Lisp as their programming language is because it's interactive. Let's type in this line of code function by function, starting at the innermost parentheses, and assign each result to a variable. By doing so we can easily learn what the code is doing.

```
Command: (setq a (entsel))
Select object: (click on the first line we drew)
(<Entity name: 88450510> (4.88461 5.55435 0.0))
```

Here we see that `entsel` allows us to select an entity and returns a list containing the entity name and the pickpoint. We're only interested in the entity name, so let's hand the list to the `car` function:

```
Command: (setq b (car a))
<Entity name: 88450510>
```

That gets us the entity name, which we'll supply to `entget`:

```
Command: (setq c (entget b))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 3.0 5.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

Now we have the entity data, so let's call `assoc`:

```
Command: (setq d (assoc 8 c))
(8 . "0")
```

This gives us the dotted pair, which we hand to the `cdr` function to obtain the result:

```
Command: (setq layer (cdr d))
"0"
```

In short, the first line of code asks the user to select an entity and returns the name of the layer on which that entity resides. It's much easier to understand when we type it in function by function, don't you think? This is a technique we use in many places when we encounter code that's difficult to understand.

Now that we have the entity's layer, the rest is easy. The second line of code calls the standard AutoCAD LAYER command and sets the layer to the result returned on the first line.

Not only is this function exceedingly useful, but we can alter it slightly to create many different, helpful utilities. For example, we could ask the user to select an entity and type in a layer name, then move the entity to that layer. Or select two entities and move the first to whatever layer the second one is on. Or select an entity and freeze the layer it's on or every layer *except* the one that it's on. A layer thaw and refreeze command is one of the labs we'll write in the next chapter.

Our choices are boundless, limited only by our imagination. Once we write a command or function, we can often discover many similar, equally useful derivatives. This is another reason AutoLISP is such a wonderful language.

♦ ♦ ♦

To this point we have seen various tasks that can be performed based on existing entity data. An equally powerful aspect of AutoLISP is our ability to modify an entity. We can easily do this by substituting new data for the existing data with a function we saw earlier, `subst`.

Look again at the entity data for the first line we drew:

```
Command: !ed1
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 3.0 5.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

Let's change the starting point of the line from 3,5 to 11,8 (the code is split across several lines for clarity, but you needn't do so):

```
Command: (setq ed1 (subst '(10 11 8)
                          (assoc 10 ed1)
                           ed1))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 11 8) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

This says (reading back to front), "In the list that is the value of `ed1`, find the pair whose car is 10 and replace it with the list `(10 11 8)`."

We can see that the value of ed1 has indeed changed. But if we flip to the graphics screen, we also see, to our great disappointment, that the starting point of the line is still the same. All we've done so far is to alter the value of ed1. For the change to actually take place, we must tell AutoCAD to modify the database. We do this by issuing the entmod function.

entmod edata	Takes entity data that has been changed and modifies the entity database to reflect that change. It returns the argument.

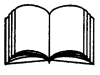
```
Command: (entmod ed1)
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 11 8) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

When we issue this function . . . Voilà! The starting point of the line is indeed changed to 11,8. If we then look at the entity data, we see that although we supplied integers for the new X and Y coordinates, AutoCAD has created a real point:

```
Command: (setq ed1 (entget e1))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 11.0 8.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

In our last substitution we used assoc because when dealing with points we're generally not sure that the entire X or Y coordinate is printed. We don't need to use assoc when we're certain of a pair's cdr. For example, the following code moves our first line to Layer 1:

```
Command: (setq ed1 (subst '(8 . "1") '(8 . "0") ed1))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "1") (100 . "AcDbLine") (10 11.0 8.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))

Command: (entmod ed1)
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "1") (100 . "AcDbLine") (10 11.0 8.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```



The use of assoc with subst is explained in Section 7.2.

We can use entmod to perform a variety of modifications, but they must make sense. For example, we can't change an entity from one type to another. On the other hand, we *can* modify entities within a block definition (however, prior to doing so, read the warning under entmod in the *AutoLISP Reference*). We can also modify subentities of a complex entity, but there's a little trick to doing so:

Imagine that we have a really intricate polyline with 200 vertices. If we want to modify every vertex, we'd probably do so in a loop that performs a `subst` to make each change, then an `entmod` to modify the database. The last thing we want to have happen is for that complex polyline to be redrawn 200 times!

Autodesk understands this and has incorporated the following condition into `entmod`: If we modify a *subentity*, `entmod` will update the database, but it won't redraw the entity. Running REDRAW won't do it either. If we do a REGEN, the drawing will reflect the changes, but it's a waste of time to REGEN for just one entity. Another entity data function, `entupd`, is used for this specific situation.

`entupd ename`	Takes the entity name of an entity or subentity and redraws the entire entity. It returns the entity name.

We earlier set `e5` to the entity name of the first polyline vertex. Now we'll access the entity data:

```
Command: !e5
<Entity name: 88450530>

Command: (setq ed5 (entget e5))
((-1 . <Entity name: 88450530>) (0 . "VERTEX") (5 . "26")
 (100 . "AcDbEntity") (67 . 0) (8 . "0") (10 1.0 1.0 0.0) . . .)
```

Next, let's change the first vertex point from 1,1 to 1,2 and `entmod` the database:

```
Command: (setq ed5 (subst '(10 1 2)
                          (assoc 10 ed5)
                           ed5))
((-1 . <Entity name: 88450530>) (0 . "VERTEX") (5 . "26")
 (100 . "AcDbEntity") (67 . 0) (8 . "0") (10 1 2) . . .)

Command: (entmod ed5)
((-1 . <Entity name: 88450530>) (0 . "VERTEX") (5 . "26")
 (100 . "AcDbEntity") (67 . 0) (8 . "0") (10 1 2) . . .)
```

Notice that a leg of the polyline has disappeared. Let's `entupd` the pline to have it redrawn with its new vertex:

```
Command: (entupd e5)
<Entity name: 88450530>
```

We can run entupd on the entity name of the polyline header or any subentity. Note that we entmod entity *data* but entupd entity *names*. Be aware, too, that we need to use entupd when we modify attributes; they're also subentities. We'll see an attribute modification example in Section 11.1.

Now we're really cooking! We've seen how to perform actions based on our entity data and even how to modify our entity data. But that's not all. AutoLISP also allows us to create entities from scratch, without using our AutoCAD drawing commands. To accomplish this feat we must first create an association list similar to the entity data we have seen, then call the entmake function to create the entity.

entmake edata	Takes an entity data list and creates the appropriate entity. It returns the list.

To create an entity we must supply the entity type and any additional information that the entity requires. Information such as layer and color is optional. If omitted, the current layer and color are used. If we supply a layer name that doesn't exist, entmake creates a new layer. Layers are unique in that respect.

The following code creates and draws a line from 10,2 to 10,4:

```
Command: (setq ed6 '((0 . "LINE") (10 10 2) (11 10 4)))
((0 . "LINE") (10 10 2) (11 10 4))

Command: (entmake ed6)
((0 . "LINE") (10 10 2) (11 10 4))
```

The new line should now be drawn on the screen. Let's look at its entity data:

```
Command: (setq ed7 (entget (entlast)))
((-1 . <Entity name: 88450590>) (0 . "LINE") (5. "29") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (10 10.0 2.0 2.59152e-144) (11 10.0 4.0 9.37315e-154) (210 0.0 0.0 1.0))
```

entmake supplies the other required information if we leave it out. The very odd Z values of the starting and ending points are due to floating point imprecision. We can safely assume that they are 0.

If we supply an entity name, entmake ignores it and creates a new one anyway. This means that we can modify an existing entity to create a new one:

```
Command: (entmake (subst '(10 9 3) (assoc 10 ed7) ed7))
((-1 . <Entity name: 88450590>) (0 . "LINE") (5 . "29") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (10 9 3) (11 10.0 4.0 9.37315e-154) (210 0.0 0.0 1.0))

Command: (setq ed8 (entget (entlast)))
((-1 . <Entity name: 88450598>) (0 . "LINE") (5 . "2A") (100 . "AcDbEntity")
(67 . 0) (8 . "0") (10 9.0 3.0 2.59152e-144) (11 10.0 4.0 9.37315e-154) (210 0.0 0.0 1.0))
```

We've now created a new line from 9,3 to 10,4. Notice that `entmake` automatically updates the entity name and handle. Observe too that we've given the new data list directly to `entmake` without first assigning to a variable.

If we supply an entity name to `entmake`, the –1 pair must come first in the entity data list; otherwise the 0 pair must be first.

♦ ♦ ♦

We can use `entmake` to create a complex entity such as a polyline. To do so we must supply data for the polyline header, each vertex, and the seqend. We could create a basic polyline header record as follows (if you're typing along, *don't* enter this line):

```
Command: (entmake '((0 . "polyline")))
```

Even the 66 field (subentities follow) is optional. However, let's be a little more daring and create a closed blue polyline from 1,6 to 3,8 to 4,7 and place it on Layer 3:

```
Command: (entmake '((0 . "polyline") (8 . "3") (62 . 5) (70 . 1)))
((0 . "polyline") (8 . "3") (62 . 5) (70 . 1))
```

Here's the meaning of these fields:

`(8 . "3")`	Put the polyline on Layer 3. If the layer doesn't exist, `entmake` creates it.
`(62 . 5)`	Make the entity blue regardless of the BYLAYER color. 62 is the group code for color and 5 is the color code for blue.
`(70 . 1)`	Make a closed polyline. The default is to create an open one.

You've probably noticed that `entmake` didn't draw anything on the screen. When we create a complex entity, AutoLISP stores each record in a temporary

file, but nothing is drawn until all of the information is supplied. We must be careful because if we make a mistake we might have to start all over again.

If we create an erroneous data record, entmake either returns nil or issues an error message. When this happens, we must run entmake with no arguments to clear the temporary file. For example, entmake will not accept the string "pline" in place of "polyline":

```
Command: (entmake '((0 . "pline") (8 . "3") (62 . 5) (70 . 1)))
nil
```

Any entmakes we do hereafter for this polyline will return nil. Therefore, we must clear the temporary file as follows and begin again:

```
Command: (entmake)
nil
```

After we build the main polyline header record, we can create the vertices:

```
Command: (entmake '((0 . "vertex") (10 1 6)))
((0 . "vertex") (10 1 6))

Command: ((entmake '((0 . "vertex") (10 3 8)))
((0 . "vertex") (10 3 8))

Command: (entmake '((0 . "vertex") (10 4 7)))
((0 . "vertex") (10 4 7))
```

Still, nothing has been drawn on the screen. We finally see the polyline when we create the seqend record:

```
Command: (entmake '((0 . "seqend")))
((0 . "seqend"))
```

We can specify vertex points on different layers or with different colors. But the main polyline entity determines the actual polyline color or layer.

In many instances where entmake might be used, we could just as easily call the command function and have AutoCAD create the entity for us. But there are situations for which entmake is quite effective. For example, suppose we read in a file containing numerous points and draw a polyline connecting them. entmake can build the pline much more quickly than the command function. Also, in Release 13, entmake can be used to create nongraphical entities such as text styles.

Another application of `entmake` is to create blocks. To build a block we must create a block header record, a record for each entity in the block, and a record of type *endblk*. As with polylines, block information is stored in a temporary file until the entire block has been specified. We might create a block comprising a circle and a line as follows:

```
Command: (entmake '((0 . "block") (2 . "test") (70 . 64) (10 0 0 0)))
((0 . "block") (2 . "test") (70 . 64) (10 0 0 0)))

Command: (entmake '((0 . "circle") (10 2 4) (40 . 1.5)))
((0 . "circle") (10 2 4) (40 . 1.5)))

Command: (entmake '((0 . "line") (10 3 3) (11 8 6)))
((0 . "line") (10 3 3) (11 8 6)))

Command: (entmake '((0 . "endblk")))
"TEST"
```

In the block header record, the 2 field is the block name. We must be careful when choosing a block name because if we specify a name that already exists, the original block definition will be replaced. If the creating function receives a block name from a user, it is good form to have it run the `tblsearch` subr to test for the block's existence prior to creating it.

The 70 field, which specifies flag bits, must be set to 64. This indicates that the block definition is referenced. Although it hasn't been referenced yet, AutoLISP assumes that it soon will be. The 10 field is the base point of the block and is typically set to `(0 0 0)`.

The endblk record doesn't appear in the block data but must be specified to denote the end of the block being built. If the final `entmake` is successful, the name of the block is returned. Thereafter the block can be inserted in our drawing with the INSERT command.

Advanced Note: When we draw a hatch pattern or an associative dimension, AutoCAD creates an *unnamed* or *anonymous* block. This is a block that cannot be directly accessed by the user. AutoLISP allows us to modify these unnamed blocks and to create our own. The procedure used in creating an anonymous block is similar to the one used to create a normal block, but the block header record is somewhat different:

```
Command: (entmake '((0 . "block") (2 . "*u") (70 . 1) (10 0 0 0)))
((0 . "block") (2 . "*u") (70 . 1) (10 0 0 0)))
```

We set a flag bit of 1 in the 70 field to indicate that this is an anonymous block. We also supply a block name of *U. When the block is built, `entmake` assigns an integer suffix to this block name. For example, our first anonymous block will be named *U0.

Because unreferenced anonymous blocks are purged when AutoCAD opens a drawing, the block name of an anonymous block might change. Thus, we must relearn an anonymous block's name each time we enter a drawing. The block name is part of the insert's entity data, which we can access via the entity's handle or by running `entsel`.

An anonymous block cannot be inserted with AutoCAD's INSERT command; it must be done with `entmake`. If we've created an anonymous block named `*U0`, we can insert it as follows:

```
Command: (entmake '((0 . "insert") (2 . "*u0") (10 0 0 0)))
((0 . "insert") (2 . "*u0") (10 0 0 0)))
```

♦ ♦ ♦

A function that is useful for very specific applications is `textbox`. We run it when we need to know the size of some text, to fit it somewhere or put other entities around it.

`textbox` edata	Takes a user-defined text entity data list and returns a list containing the diagonal coordinates of a box that would enclose that text.

We can supply text directly to `textbox` in the following manner (1 is the group code for text):

```
Command: (textbox '((1 . "Parts List")))
((0.0 -3.46945e-017 0.0) (1.86667 0.2 0.0))
```

Alternatively, we can supply existing text entity data. Let's create some text:

```
Command: (command "text" '(9 5) "" "" "Schematic A")
nil

Command: (textbox (entget (entlast)))
((0.0 0.0 0.0) (2.06667 0.2 0.0))
```

The first point returned is the lower left corner of the text. It is generally at or near the point (0.0 0.0 0.0). The other point is the upper right corner. These must be added to the text insertion point in order to get the true text location.

The *AutoLISP Reference* explains that the beginning coordinates will vary from 0,0 if the text is "oblique or vertical, or it contains letters with descenders (such as *g* and *p*)." The fact that one or more of these numbers might be negative doesn't affect the way that we use them.

`textbox` measures the text using the current text style. However, it ignores text rotation and returns corner points as though the text is horizontal.

Example

The *AutoLISP Reference* shows a function called `c:tbox` that draws a textbox around any text, regardless of its orientation. The following modification to that function adds a border so that the box doesn't touch the text.

```
   (defun c:tbox (/ en tb)
1    (setq en (car (entsel "Select text: ")))
     (command "ucs" "entity" en)                    ;Set UCS to starting pt of text
2    (setq tb (textbox (entget en)))                ;Determine corners of textbox
     (boxit (car tb) (cadr tb) en)                  ; and draw box
     (command "ucs" "p")                            ;Restore previous UCS
     (princ))

3  (defun boxit (p1 p2 en / off)
     (setq p1 (trans p1 1 en)                       ;Translate points from UCS to ECS
           p2 (trans p2 1 en))
     (setq off (getdist "\nEnter offset for border: "))
4    (entmake '((0 . "polyline") (70 . 1)))        ;Make a polyline box around text
     (entmake (list '(0 . "vertex") (list 10 (- (car p1) off) (- (cadr p1) off))))
     (entmake (list '(0 . "vertex") (list 10 (- (car p1) off) (+ (cadr p2) off))))
     (entmake (list '(0 . "vertex") (list 10 (+ (car p2) off) (+ (cadr p2) off))))
     (entmake (list '(0 . "vertex") (list 10 (+ (car p2) off) (- (cadr p1) off))))
     (entmake '((0 . "seqend"))))
```

1. The `c:tbox` command asks the user to select a text entity and sets the variable `en` to the entity name. It adopts the UCS of that entity.

2. Next, it gets the entity data and gives it to the `textbox` function. `textbox` returns the lower left and upper right coordinates of the text itself. These coordinates are supplied as arguments to the `boxit` function. When control comes back to the `c:tbox` command, it restores the previous UCS and exits.

3. `boxit` has `trans` translate each point from the current UCS (as indicated by the 1) to the entity coordinate system of the text it will box. It then re-

quests an offset for the border. If we're going to use this function numerous times and we always want the offset to be, say, 0.2, we can hard code the offset into the function and not bother asking the user each time. But as it's now structured the function is more flexible.

4. `boxit` draws the box by having `entmake` build a closed polyline. For each vertex it adds or subtracts the offset from the X and Y coordinate in order to provide for the border. Although the box could also be drawn by simply calling the PLINE command, `entmake` runs faster. A version of the `boxit` function using PLINE is shown in Section 13.1.2.

10.4 Handles

Handles provide another means of accessing entities. We can use them to circumvent a problem associated with grouping entities into selection sets. If you're just starting to learn AutoLISP, you might want to defer this topic until you've studied selection sets in the next chapter.

Suppose we have a drawing comprising hundreds of entities, and we've built a selection set of, say, 25 entity names on which we're performing various operations. When our drawing session is over, we want to save this collection of entities so that we don't have to select them again when we reenter the drawing. Ideally, we'd like to write the entity names to a file, then read the file back in the next time we edit the drawing.

Unfortunately, we can't do this because the entity names aren't constant; an entity is not guaranteed to have the same entity name each time we enter our drawing. Let's say that the first three entities we create are a line, a circle, and an arc and that they have the following entity names:

```
line    <Entity name: 88450510>
circle  <Entity name: 88450518>
arc     <Entity name: 88450520>
```

During the course of the present drawing session, we delete the line. The next time we enter this drawing the entities will have the following entity names:

```
circle  <Entity name: 88450510>
arc     <Entity name: 88450518>
```

Since the circle is now the first entity in the database, its entity name is `<Entity name: 88450510>`. If we had written its original entity name to a file, we'd now access the arc!

We overcome this problem by using structures called *handles*. Handles are unique strings of hexadecimal numbers, such as "29", "2A", "2B", that are assigned sequentially to each existing and new entity (and subentity) in the database. They are part of the entities' data and have a group code of 5. Because an entity's handle never changes, we can use it with 100 percent confidence that we are always accessing the correct entity.

Release 12 Note

Starting with Release 13, handles are always present in each entity's data. Prior AutoCAD versions require that we turn this facility on with the HANDLES command, as follows:

```
Command: handles
Handles are disabled.
ON/DESTROY: on
```

Once we turn on handles they usually stay for the life of the drawing. When the drawing is finished, we can run the HANDLES command again to destroy the handles so that the drawing is archived in a smaller space.

When we choose DESTROY, HANDLES prints a long paragraph on the screen and specifies a password for us to type. The password changes each time so that we can't write a function to destroy handles; we must do it at the Command: prompt.

♦ ♦ ♦

If we want to be able to reconstruct a selection set the next time we enter this drawing, we store the appropriate handles in an external database prior to ending the present drawing session. When we return to our drawing, we can access each entity via its handle by using AutoLISP's `handent` function.

`handent str`	Takes an entity's handle and returns its entity name.

Looking again at the data for the first line we drew we notice that there's a dotted pair with a 5 group code:

```
Command: (setq ed1 (entget e1))
((-1. <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "1") (100 . "AcDbLine") (10 11.0 8.0 0.0) (11 6.0 6.0 0.0) (210 0.0 0.0 1.0))
```

We extract the entity's handle like so:

```
Command: (setq eh1 (cdr (assoc 5 ed1)))
"22"
```

Now we can write this handle to a file and read it back the next time we edit our drawing. Once we have the handle, `handent` can locate the entity name:

```
Command: (handent eh1)
<Entity name: 88450510>
```

The entity name might be different from the last session, but the handle stays the same.

Incidentally, if we know the handle of a particular entity, we can use `handent` as input to an AutoCAD command without bothering to learn the entity name. For example, we can copy an entity as follows:

```
Command: (command "copy" (handent eh1) "" pause pause)
nil
```

10.5 Advanced Topic: Extended Entity Data (Xdata)

Sometimes an application needs to place its own information with a particular entity. AutoLISP allows us to store up to 16K of data with each entity that we create.

The extended entity data facility was initially created to allow third-party software developers to store necessary information with entities they need to access or modify. But this is a facility that we can use to our own advantage as well.

For example, suppose that we have a polyline representing a sheet metal object. The metal might have different thicknesses in different parts or have a different composition. Extended entity data lets us put the appropriate information with each vertex of the polyline. Yes, we can do this with attributes, but extended entity data is more versatile and needn't be attached to a block.

Xdata is also useful when several people might update the same drawing. The reviser can put his or her initials, a revision number, and the current date along with the modified entity. Furthermore, extended entity data never appear in the drawing itself, so the facility can be used to store any information we want hidden from someone using our drawing.

We can access extended entity data with `entget`. But if several different applications have written Xdata to the same entity, we usually only want to retrieve our own data. We must have some means of keeping various applications' Xdata separate and distinct. This is done with an application ID.

An *application ID* (also called an *application name*) is a unique string that identifies our application so that no one will accidentally access or alter our Xdata (and so that we won't unknowingly modify anyone else's). Suitable choices might be your initials or birthday appended to the product name. Our first task when working with extended entity data is to choose an application ID and *register* our application.

`regapp` str	Registers the extended entity application ID supplied as an argument. It returns the application ID.

Let's begin a new drawing and register a couple of applications (it will aid your understanding if you enter the following code as you read):

```
Command: (regapp "rh-rev")
"RH-REV"

Command: (regapp "rh-mass")
"RH-MASS"
```

`regapp` returns the name of the application as an uppercase string. If the application is already registered, `regapp` returns `nil`. The application ID can be up to 31 characters in length but *cannot* contain spaces.

Now draw a line from 4,2 to 8,4 and make its entity data be the value of `ed1`:

```
Command: !ed1
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 4.0 2.0 0.0) (11 8.0 4.0 0.0) (210 0.0 0.0 1.0))
```

Our next step is to create the extended entity data itself. The `"rh-rev"` application will contain the date and a description of the drawing revision. `"rh-mass"` will contain a real number that denotes the mass of the surface of the object on which we've drawn the line. We use group code 1000 to indicate a string and 1040 to indicate a real number.

```
Command: (setq rev '("rh-rev" (1000 . "3-14-95")
                              (1000 . "Altered thickness")))
("rh-rev" (1000 . "3-14-95") (1000. "Altered thickness"))

Command: (setq mass '("rh-mass" (1040 . 1.45)))
("rh-mass" (1040 . 1.45))
```

Each extended entity data record is a list whose car is the application ID. The remaining list elements are dotted pairs whose car is an extended entity group code and whose cdr is the value for that group code.

We must now merge both of these Xdata records into one list that can be appended to the existing entity data. Extended entity data always begin with a –3 group code. For clarity in these examples we'll format the long entity data lists that are returned by the functions.

```
Command: (setq app (list -3 rev mass))
(-3
 ("rh-rev" (1000 . "3-14-95") (1000 . "Altered thickness"))
 ("rh-mass" (1040 . 1.45)))
```

As this example shows, extended entity data is a list whose car is –3 and whose other elements are extended entity records. Now that we've created the Xdata, we can append it to the existing entity data and call `entmod` to modify the database:

```
Command: (entmod (append ed1 (list app)))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 4.0 2.0 0.0) (11 8.0 4.0 0.0) (210 0.0 0.0 1.0)
 (-3 ("rh-rev" (1000 . "3-14-95") (1000 . "Altered thickness"))
     ("rh-mass" (1040 . 1.45))))
```

It may seem odd that we have to say `(list app)`, but remember that `append` merges lists into one long list. Unless we make a list of the Xdata list, `append` will strip off a needed set of parentheses, and we'll subsequently get an error. This is analogous to taking the lists `(a b c)` and `(1 2 3)` and creating the new list `(a b c (1 2 3))`. We'd accomplish this with the following code:

```
Command: (append '(a b c) (list '(1 2 3)))
(A B C (1 2 3))
```

♦ ♦ ♦

Occasionally we may have a need to update an Xdata record that we've already added to an entity. To add a string to the `"rh-mass"` record, we first append the string pair to the existing record. Note that we again need the `list` function for the reason that was just explained.

```
Command: (setq mass2 (append mass (list '(1000 . "Center thicker"))))
("rh-mass" (1040 . 1.45) (1000 . "Center thicker"))
```

Next, we add the –3 group code, append this list to `ed1`, and `entmod`:

```
Command: (setq app2 (list -3 mass2))
(-3 ("rh-mass" (1040 . 1.45) (1000 . "Center thicker")))

Command: (entmod (append ed1 (list app2)))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 4.0 2.0 0.0) (11 8.0 4.0 0.0) (210 0.0 0.0 1.0)
 (-3 ("rh-mass" (1040 . 1.45) (1000 . "Center thicker"))))
```

The Xdata facility is smart enough to replace the original `"rh-mass"` application with this new one.

We can add an entirely new extended entity application in a similar manner. Register the application, create the extended entity data (including the –3 group code), append it to the normal entity data, and `entmod`. The Xdata facility will merge the new application without destroying any existing data.

Once the database has been updated we can access the extended entity data with `entget`. We supply to `entget` an optional second argument, which is a list containing the names of the applications we want returned with the main entity data:

```
Command: (entget e1 '("rh-rev"))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 4.0 2.0 0.0) (11 8.0 4.0 0.0) (210 0.0 0.0 1.0)
 (-3 ("RH-REV" (1000 . "3-14-95") (1000 . "Altered thickness"))))
```

We can retrieve as many Xdata applications as we choose:

```
Command: (entget e1 '("rh-rev" "rh-mass"))
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 4.0 2.0 0.0) (11 8.0 4.0 0.0) (210 0.0 0.0 1.0)
 (-3 ("RH-MASS" (1040 . 1.45) (1000 . "Center thicker"))
     ("RH-REV" (1000 . "3-14-95") (1000 . "Altered thickness"))))
```

We can still retrieve the entity data without any extended data:

```
Command: (entget e1)
((-1 . <Entity name: 88450510>) (0 . "LINE") (5 . "22") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 4.0 2.0 0.0) (11 8.0 4.0 0.0) (210 0.0 0.0 1.0))
```

Note that we append Xdata to the entity *data*, but when we retrieve it we supply the entity *name* to `entget`.

Extended entity data space is limited to 16,383 bytes per entity. Although that sounds like a large amount, we could conceivably run out of room. Thus, two additional functions exist that allow us to determine whether our application is too large.

`xdroom` ename	Returns the amount of extended entity space that is currently available for the specified entity.

```
Command: (xdroom e1)
16324
```

`xdsize` list	Takes an Xdata application list and returns the number of bytes that it will consume when appended to the entity data.

```
Command: (xdsize app)
43

Command: (xdsize app2)
28
```

We can use `xdsize` in conjunction with `xdroom` to determine whether an application will fit before trying to append it.

Example

The following command requests entity data and extended entity data information from the user, then adds the Xdata to the entity data. Assign to `e2` and `ed2` the entity name and the entity data, respectively, for a line from 3,6 to 7,2. Then run the `c:xdadd` command as follows:

```
Command: xdadd
Entity data to modify: ed2
Application ID: rh-test
Type of data to add (<String> Real): r
Enter real number: 55.6

Command: (entget e2 '("rh-test"))
((-1 . <Entity name: 88450518>) (0 . "LINE") (5 . "23") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 3.0 6.0 0.0) (11 7.0 2.0 0.0) (210 0.0 0.0 1.0)
 (-3 ("RH-TEST" (1040 . 55.6))))
```

Now let's look at the code:

```
  (defun-c:xdadd (/ ed id type data app)
1   (setq ed (eval (read (getstring "Entity data to modify:")))
        id (getstring "Application ID: "))
    (regapp id)                                          ;Register application
    (initget "String Real")                              ;Limit input
2   (setq typ (getkword "Type of data to add (<String>Real): "))
    (if (eq typ "Real")                                  ;User want real?
        (setq data (cons 1040 (getreal "Enter real number: ")))   ;Yes, get data
        (setq data (cons 1000 (getstring t "Enter string: "))))   ;No, get string
3   (setq app (list -3 (list id data)))                  ;Build Xdata
4   (entmod (append ed (list app)))                      ;Add it onto entity
    (princ))
```

1. Most of this routine consists of setting variables. It uses `getstring` to input the symbol whose value is the entity data. Since the argument is a string, for example, `"ed2"`, `c:xdadd` must convert it to a symbol (with `read`), then explicitly `eval` the symbol to get the entity data itself. It then gets the application ID and registers the application.

2. `getkword` inputs the user's choice of object type. Thereafter the appropriate group code and the data entered by the user are consed together to form a new dotted pair. We could easily expand this code to provide for integers, points, or one of several other datatypes.

3. The last two lines (excepting the `princ`) are the most interesting for our study of extended entities. These are where the Xdata itself is built. The innermost function on line 3 builds a list of the application ID and the data, such as

   ```
   ("RH-TEST" (1040 . 55.6))
   ```

   –3 and this list are then handed to the `list` function, and the list that is returned is assigned to `app`:

   ```
   (-3 ("RH-TEST" (1040 . 55.6)))
   ```

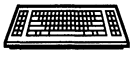
4. The next line builds a list whose only element is the list created in step 3 and appends it onto the entity data itself. It then `entmod`s the entity data to add the extended data to it.

10.6 Advanced Topic: Accessing Entities within Blocks

We use `nentsel` to access nested entities, and most of the time the results it returns are similar to those returned by `entsel`. But when the user selects an entity nested in a block insert, `nentsel` returns a list consisting of the entity name and the pickpoint, plus two additional pieces of information:

- A *Model to World Transformation Matrix.* This is a list of coordinates that enables us to translate the supplied set of points from the Model Coordinate System (in which the points are expressed) to the World Coordinate System (which we can understand).

- A list containing the entity name of the block of which the selected entity is a part. If the insert itself is nested in another block, the list contains all of the block entity names, beginning with the innermost block.



The matrix is most easily understood if we look at an example. Set up your environment as follows:

- In a fresh drawing create a line from 1,2 to 4,3.

- Build a block called A containing only that line, with the insertion base point at 1,2.

- Insert Block A exactly where the line was originally, starting at 1,2.

- Draw a small circle near the line, then create a new block, B. It should consist of both the circle and the original block, with the insertion base point again at 1,2.

- Insert Block B at the point 8,3.

Next, run `nentsel` and select a point on the line in the latest insert:

```
Command: (setq e (nentsel))
Select object: (click on line)
(<Entity name: 88450530> (9.15658 3.30517 0.0)
 ((1.0 0.0 0.0) (0.0 1.0 0.0) (0.0 0.0 1.0) (8.0 3.0 0.0))
 (<Entity name: 88450558> <Entity name: 88450570>))
```

The information returned is a list containing the entity name of the line, the pickpoint, the Model to World Transformation Matrix, and a list containing the entity name of Block A followed by the entity name of Block B.

We can access the entity data for the selected line in the following manner:

```
Command: (entget (car e))
((-1 . <Entity name: 88450528>) (0 . "LINE") (5 . "26") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbLine") (10 0.0 0.0 0.0) (11 3.0 1.0 0.0) (210 0.0 0.0 1.0))
```

Notice that the starting and ending points of the line are listed as `(0.0 0.0 0.0)` and `(3.0 1.0 0.0)`, respectively. These aren't the actual points; they must be transformed. Let's translate the ending point (the 11 field).

The Transformation Matrix translates points from the Model Coordinate System to the World Coordinate System according to the following formula:

$$X_{wcs} = (X1 * X_{mcs}) + (X2 * Y_{mcs}) + (X3 * Z_{mcs}) + X4$$
$$Y_{wcs} = (Y1 * X_{mcs}) + (Y2 * Y_{mcs}) + (Y3 * Z_{mcs}) + Y4$$
$$Z_{wcs} = (Z1 * X_{mcs}) + (Z2 * Y_{mcs}) + (Z3 * Z_{mcs}) + Z4$$

In this formula, X_{mcs}, Y_{mcs}, and Z_{mcs} are the coordinates of the point we plan to translate from the Model Coordinate System, in this case `(3.0 1.0 0.0)`.

X1-4, Y1-4, and Z1-4 are the X, Y, and Z coordinates in the four lists returned by `nentsel`. Thus,

```
XYZ1 = (1.0 0.0 0.0)
XYZ2 = (0.0 1.0 0.0)
XYZ3 = (0.0 0.0 1.0)
XYZ4 = (8.0 3.0 0.0)
```

We'll substitute these numbers into the formula:

$$X_{wcs} = (1.0 * 3.0) + (0.0 * 1.0) + (0.0 * 0.0) + 8.0) = 11.0$$
$$Y_{wcs} = (0.0 * 3.0) + (1.0 * 1.0) + (0.0 * 0.0) + 3.0) = 4.0$$
$$Z_{wcs} = (0.0 * 3.0) + (0.0 * 1.0) + (1.0 * 0.0) + 0.0) = 0.0$$

This gives us a result of `(11.0 4.0 0.0)`, which is the endpoint of the line as it appears on the screen.

If we step back a bit from the formula and look at the whole picture, we can see what's happening. The line's coordinates are returned as though its starting point is 0,0; the matrix is a tool for translating each point to where it should be in the drawing. Through standard matrix multiplication techniques we could scale or rotate the block as well.

When accessing entities in a block, `nentselp` uses a 4 × 4 transformation matrix. The last row and column of `nentselp`'s transformation matrix is ignored, making it effectively identical to `nentsel`'s.

10.7 Important Points to Remember

- You'll notice that all of the entity function names begin with an *ent* prefix, which can get a little confusing. The suffix clearly specifies the task that the function performs.

- `(entnext (entlast))` usually returns `nil`. But if the last main entity is complex, it returns the entity name of the first subentity.

- AutoLISP has no conventions for labeling entity names and entity data. This book's convention is to make entity names the values of the symbols `e1`, `e2`, `e3`, and so on, and make the corresponding entity data the values of the symbols `ed1`, `ed2`, `ed3`, and so on. Lab Exercise 9 has us write a command to perform these assignments automatically.

  If you have a system you like better; that's fine. But please adopt a naming convention and adhere to it. Otherwise you'll forever confuse entity names and entity data.

- When modifying a subentity, we must remember to `entmod` the entity *data*, then `entupd` the entity *name*. Similarly, we add Xdata by appending it to an entity's *data* and running `entmod`, then read it by running `entget` on the entity's *name*.

10.8 Labs

1. Load `examples.lsp` and run the command `ent-set`. Answer the following questions using AutoLISP to perform the indicated steps. This exercise should be done at the Command: prompt.

   a) What are the endpoints of each red line?

   b) What layer is each of the red lines on?

   c) Use entity functions to locate the centers of the circles, then use `entmake` to create a line between them. This can be tricky; don't try to do too much in one step.

   d) Now delete this line.

   e) Change the right-most line segment so that the red lines become a closed triangle.

   f) Alter the box to have half its current width.

2. `entlast` finds the last main entity in the database. Write a function that returns the last entity, be it a main or subentity, in the database.

3. We constantly use `subst` with `assoc` to replace one entity data pair with another, thereby changing the entity. Write a function that takes a group code, a new value, and an entity datum as arguments and performs the substitution for us. For example, instead of having to enter

   ```
   Command: (setq ed5 (entmod (subst '(10 3 3) (assoc 10 ed5) ed5)))
   ```

   we can now enter

   ```
   Command: (setq ed5 (sub 10 '(3 3) ed5))
   ```

4. Write a command called `c:nl` that prompts the user for an entity and layer name, then moves the selected entity to the chosen layer.

5. Write a command that prompts the user to select a color and one or more entities, then changes the selected entities' color (the SELECT and CHANGE commands come in handy here).

   Optional: Make one choice "BYENTITY" so the user can make an entity's color the same as that of another entity. This may require the `tblsearch` function.

```
Command: cc
Red/Yellow/Green/Cyan/Blue/Magenta/White/byLayer/byEntity: r
Select item(s) to change color of:
```

   The user selects 1 or more entities, and they are changed to red.

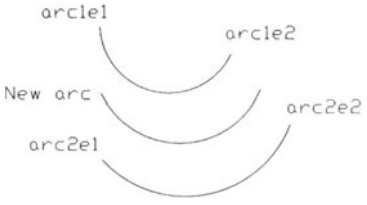
```
Command: cc
Red/Yellow/Green/Cyan/Blue/Magenta/White/byLayer/byEntity: e
Select item(s) to change color of: (Select 1 or more entities)
Pick entity with desired color for matching:
```

   When the user selects the red entity, the chosen entities are changed to red.

6. In Section 9.3 we wrote a command called `c:midln` to draw a line midway between two other lines. Modify that command to use entity rather than measurement functions, and rename it `c:midln2`. Also, highlight the first line selected (the command runs too quickly to make highlighting the second line worthwhile) and unhighlight it when the function finishes.

   A way to approach this is to use `entget` to access the entity data. Verify that the user indeed selected lines, then read the entity database to learn the starting and ending points of each line. You don't have to modify the latter portion of the command where the midline is actually calculated and drawn.

7. Write a command named `c:midarc` that works like the `c:midln2` command that we wrote in Lab Exercise 6 except that it draws an arc midway between two other arcs. The starting point of the new arc should be midway between the starting points of the other two arcs, and the ending point should be midway between the other arcs' ending points. The following picture shows two arcs with the midarc drawn between them:



   This exercise requires you to access arc group codes 10 (center point), 40 (radius), 50 (starting angle), and 51 (ending angle).

8. Write a function that accepts a list of points as an argument and uses `entmake` to create a closed polyline connecting the points.

9. Write a command called `c:ents` to label the database of existing entity names `e1`, `e2`, and so on. It should also name the associated entity data `ed1`, `ed2`, and so forth. Thus, for N entities, there should be variables `e1` to `eN` and `ed1` to `edN` bound to the entity names and entity data, respectively.

   This exercise assumes you have some familiarity with the creation and manipulation of selection sets, which are covered in the next chapter. Also, your solution may require explicit use of `read`, `set`, and `eval`.

10. Write a function that takes a selection set as an argument, creates a list of handles for all entity names in that selection set, then writes that list to a file. You might choose the drawing name as the default filename (change the extension), or use `getfiled` to request the filename from the user. If you're running Release 12, you must turn on the handles facility before running this function.

```
Command: (handout ss1)
File to write: my-dwg.dat
```

    When the handout function runs correctly, write another function that reads the file, accesses the handles, and returns a selection set containing the appropriate entity names. This exercise can be written using `mapcar` and `lambda` but doesn't have to be.

```
Command: (setq ss2 (handin))
File to read: my-dwg.dat
```

11. In Section 10.5 we saw the `c:xdadd` command, which requests information from the user and adds the supplied extended entity data to the specified entity. This function has a problem: If the user provides an application ID that is already present in the Xdata, the command overwrites the existing extended entity data with the new information instead of adding it on.

    Write a command (or alter `c:xdadd`) that asks the user to select an entity and type in Xdata, then adds the new extended entity data without overwriting any existing Xdata. For example, suppose an entity already has the extended data `(-3 ("RH-TEST" (1040 . 55.6)))`. We can run `c:xdalt` to add string data as follows:

```
Command: xdalt
Select entity to modify: (select entity)
Application ID: rh-test
Type of data to add (<String> Real): s
Enter string: abcde

Command: (entget e2 '("rh-test"))
((-1 . <Entity name: 88450518>) (0 . "LINE") (5 . "23") (100 . "AcDbEntity") (61 . 0)
 (8 . "0") (100 . "AcDbLine") (10 3.0 6.0 0.0) (11 7.0 2.0 0.0) (210 0.0 0.0 1.0)
 (-3 ("RH-TEST" (1040 . 55.6) (1000 . "abcde"))))
```

    Hint: Get the existing extended entity data from the entity and append the new Xdata to it.

12. If you work with blocks, you'll probably find this exercise quite useful. It addresses the occasional need to explode a block insert to alter an entity contained within. In a large drawing it's difficult to subsequently locate the entities to reblock them.

    Write a command that saves the entity name of the last entity in the drawing, then explodes a block selected by the user. All of the entities that follow the former `entlast` in the entity database were members of the exploded block. Tag these entities with extended entity data and record their application ID in a global variable.

    Write a second command that builds a selection set of all entities having the saved application ID, then calls the BLOCK and INSERT commands to reestablish the original block insert.

    This exercise is a bit lengthy but not that hard if you understand how to create and access Xdata. However, it also requires a basic knowledge of selection sets, which are presented in the next chapter. The Advanced Note on extended entity filtering in Section 11.2.1 provides a hint that might simplify the program.

11

Selection Sets and Nongraphical Entities

Topics Covered in This Chapter

- Building selection sets
- Manipulating entities in selection sets
- Accessing and altering nongraphical entities

Goals for This Chapter

- Learn how to group entities into selection sets and to add and delete entities from selection sets.
- Understand how judicious use of selection sets gives us control over our drawings.
- Learn how AutoLISP routines can be applied to general data sets or specific entities in our drawings.
- Discover how we access and manipulate AutoCAD table and dictionary information. Learn how to have code conditionally executed based on this information.

Introduction

An outstanding advantage of using AutoLISP lies in our ability to manipulate selection sets. A *selection set* is a collection of entity names that we've grouped together for some purpose. A selection set is a new AutoLISP object type, the last one we'll encounter. It has the following printed representation:

```
<Selection set: 1>
```

Selection sets are actually very familiar to us; we create and use them all the time. When we select objects for a COPY or MOVE command, a selection set is built. The difference is that in AutoCAD we use a selection set once (or occasionally several times by invoking the Previous option), whereas in AutoLISP we give the selection set a name (via a symbol binding) and use it over and over again. Furthermore, our ability to easily access and alter the entity data makes selection sets more versatile than groups.

For example, we might create a selection set of the holes we drilled through a plate. Whenever we create a new hole, we can have it automatically added to that selection set. Alternatively, when we're done drawing the holes, we can write a simple function to add all or some of them to a selection set. Thereafter we can easily move all holes to their own layer, increase their diameters, or alter their locations on the screen.

Like entity names, selection sets are valid input to AutoCAD's Select objects: prompt. We can supply an existing selection set to the MOVE command rather than manually selecting the objects on the screen.

AutoCAD uses tables and dictionaries to store information about blocks, layers, and other structures. Functions for manipulating these nongraphical entities are also examined in this chapter.

11.1 Basic Selection Set Functions

This section introduces the "meat and potato" functions that we use to create and manipulate selection sets. The examples shown are easier to follow if you enter them at the Command: prompt.

ssadd [ename] [ss]	Creates a new selection set or adds an entity name to an existing selection set. It returns the selection set.

ssadd can take zero, one, or two arguments.

- If called with no arguments, ssadd creates a new selection set with no members.
- If called with an entity name, ssadd creates a new selection set with that entity name as its sole member.
- If called with an entity name and a selection set, ssadd adds that entity name to the existing selection set.

Let's start a new drawing and create a few entities:

```
Command: (command "line" '(1 5) '(3 2) '(4 6) "")
nil

Command: (setq e1 (entnext))                    ;First line
<Entity name: 88450510>

Command: (setq e2 (entlast)                     ;Second line
<Entity name: 88450518>

Command: (command "circle" '(8 2) 1.0)
nil

Command: (setq e3 (entlast))                    ;circle
<Entity name: 88450520>
```

Next we'll create a couple of selection sets.

```
Command: (setq ss1 (ssadd))                     ;Create an empty ss
<Selection set: 1>

Command: (setq ss2 (ssadd e1))                  ;Create new ss
<Selection set: 2>                              ; containing e1
```

We generally use ssadd with no arguments or one argument as an initialization step prior to entering a loop. Naturally, a selection set containing zero or even one entity name isn't much use to us. But once the selection set is created, we can add to it as follows:

```
Command: (ssadd e2 ss2)                    ;Add e2 to ss2
<Selection set: 2>
```

We don't have to add entity names to a selection set within a loop; that's just a frequent practice. We can actually add to selection sets in any manner we choose, such as:

```
Command: (ssadd (entnext e2) ss2)          ;Add 3rd entity name
<Selection set: 2>                         ; to ss2
```

Once we've created a selection set, we can supply it to any AutoCAD command that asks us to select objects:

```
Command: mirror
Select object: !ss2 3 found
```

All three entities in the selection set are highlighted.

Here's a word of warning regarding the printed representation of selection sets. Unlike other object types, the printed representation of a selection set contains a *decimal* number. This number is assigned sequentially and tells us nothing about how many or which entity names the selection set contains. In fact, the printed representation can actually confuse us.

An AutoLISP convention is to name our selection sets `ss1`, `ss2`, and so forth. Because selection sets are created in AutoCAD as well as in AutoLISP, the number we use to designate a particular selection set might be different from the number in its printed representation, like so:

```
Command: !ss5
<Selection set: 8>
```

This can lead to our entering `ss8` when we really mean `ss5`, which of course will return an erroneous result. Our safest approach is to ignore a selection set's printed representation.

We can determine the number of entity names a selection set contains with the `sslength` function.

`sslength ss`	Returns the number of entity names in the selection set.

```
Command: (sslength ss2)
3
```

```
Command: (sslength ss1)
0
```

The number is returned as an integer unless it's greater than 32767, in which case it's returned as a real number.

As in AutoCAD proper, when we attempt to add an entity name that is already in a selection set, it isn't actually added a second time:

```
Command: (ssadd e2 ss2)                ;e2 is already in ss2
<Selection set: 2>

Command: (sslength ss2)                ;Thus ss2's length remains the same
3
```

We can remove an entity name from a selection set with `ssdel`.

`ssdel` ename ss	Deletes the entity name from the selection set. It returns the selection set or `nil` if the entity is not found.

```
Command: (ssdel e1 ss2)                ;Returns ss2 with e1 removed
<Selection set: 2>

Command: (sslength ss2)                ;Now ss2 contains only 2 enames
2

Command: (ssdel e1 ss2)                ;e1 isn't in ss2
nil
```

Once an entity name has been removed from a selection set, we can use `ssadd` to put it back in.

```
Command: (ssadd e1 ss2)                ;Returns e1 to the selection set
<Selection set: 2>
```

Since entity names are always added to the end of a selection set, `e1` is not in the same position that it was originally. However, the order of entity names in a selection set doesn't usually matter.

ssmemb ename ss	Returns the entity name if it is a member of the selection set and nil if it isn't.

```
Command: (ssmemb e2 ss2)              ;Returns e2's entity name
<Entity Name: 88450518>

Command: (ssmemb e2 ss1)              ;e2 is not a member of ss1
nil
```

If we erase an entity that is a member of a selection set, it stays in the selection set; ssmemb will find it. However, actions performed on the selection set have no effect on the deleted entity.

ssname ss n	Returns the *nth* entity name in the selection set, starting with position 0.

Recall the nth function that we studied in Chapter 7. The behavior of ssname is quite analogous except that it operates on selection sets instead of lists. Also, in nth the index is the first argument, whereas in ssname it is specified second.

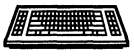
```
Command: (ssname ss2 0)               ;First entry, e2
<Entity name: 88450518>

Command: (ssname ss2 1)               ;Second entry, e3
<Entity name: 88450520>

Command: (ssname ss2 2)               ;Third entry, e1
<Entity name: 88450510>
```

Since we deleted the first line we drew from the selection set and readded it to the end, its entity name is encountered last. Keep in mind that the elements are numbered counting from 0. Thus,

```
Command: (ssname ss (sslength ss))   ;sslength is 3
nil
```

Example

If you have the need to access or manipulate attributes of blocks, the following command provides a starting point. It makes a selection set of all attributes in the drawing, successively encircles each attribute's text, and prompts the user for a new attribute value.



Prior to running this command, clear off your screen and run the `c:att-set` command that is in the `examples.lsp` file:

```
Command: att-set
```

`c:att-set` creates a block consisting of a box and two attributes, then inserts the block into the drawing twice. Let's run the `c:att-ed` command:

```
Command: att-ed
Enter new attribute value:
```

`c:att-ed` draws a circle around an attribute and prompts for a new value. If we supply one, the original attribute text is replaced. If we press <enter>, the original text remains. This process is repeated for every attribute in the drawing. Let's look at the code:

```
1 (defun c:att-ed (/ e ed ss cnt str)
    (setq e  (entnext)                                  ;Get first ename
          ss (ssadd)                                    ;Create null ss
          cnt 0)                                        ;Index into ss for ssname
2   (while (setq e (entnext e))                         ;Get next entity
      (if (eq (cdr (assoc 0 (entget e))) "ATTRIB")      ;Is it an attribute?
          (ssadd e ss)))                                ;Yes, add it to ss
3   (while (setq e (ssname ss cnt))                     ;Sequence until end of ss
      (setq ed  (entget e)                              ;Get entity data
            cnt (1+ cnt))                               ;Inc index to next ename in ss
      (command "circle" (cdr (assoc 10 ed)) 0.7)        ;Encircle data
4     (setq str (getstring t "\nEnter new attribute value: "))
      (if (not (equal str ""))                          ;User didn't hit <enter>
          (progn
            (setq ed (subst (cons 1 str)                ;Change to new value
                            (assoc 1 ed) ed))
            (entmod ed)                                 ;Update database
            (entupd e)))
      (entdel (entlast))                                ;Remove circle
    )       ;end while
    (prin1))
```

1. The `c:att-ed` command starts by initializing `e` to the first entity name in the database, `ss` to an empty selection set, and the selection set index, `cnt`, to 0. These are standard steps that we frequently perform prior to entering a loop.

2. The first `while` loop accesses each entity name in turn and examines its 0 field to see whether it is an attribute. If it is, the entity name is `ssadd`ed to the selection set. Thus, after all entities have been examined, the selection set contains the entity names of every attribute in the drawing.

The test to the `while` is to get the next entity name in the entity database. `entnext` returns `nil` when the end of the database is reached, and the `while` automatically exits. You might have observed that the very first entity is never examined. That's OK because an attribute is a subentity and therefore can never be the first entity in the database.

3. The second `while` loop accesses each entity name in the selection set. Remember that `ssname` is like `nth` in that it begins counting from 0. When the end of the selection set is reached, `ssname` returns `nil` and the `while` automatically exits.

   Within the loop, `ed` gets bound to the entity data for the attribute and `cnt` is incremented. A circle of radius .7 is drawn centered at the starting point of the attribute text, which is the 10 field in the attribute data. Since the circle's existence is temporary and used only to identify a particular attribute, no attempt is made to ensure that the circle entirely surrounds the attribute text.

4. `getstring` requests a new attribute value and the entered text is assigned to the symbol `str`. If the user presses `<enter>`, `str` is bound to the null string, `""`.

   The `if` tests whether the user did indeed enter the null string. This is where the `not` function comes into play. We use it when we want an `if` to take the THEN path if its test *fails*. If the value of `str` is *not* the null string, the existing attribute text is replaced with the string that the user just entered. A group code of 1 always specifies text.

   Finally, the entity database is modified with `entmod`, the attribute text is redrawn with `entupd`, and the circle is deleted. The loop is then repeated for the remaining entity names in the selection set.

You might be aware that the AutoCAD ATTEDIT command can accomplish the same task as `c:att-ed`. However, this command is convenient to use and it gives us access to the attributes' entity data. It can also be modified to perform numerous different actions on attributes.

11.2 Using `ssget` to Build Custom Selection Sets

The `ssget` function is the most powerful and versatile mechanism for building selection sets. It allows us to create or add to selection sets by either using our AutoCAD pointing device to select multiple objects in a drawing or by specifying entities based on certain criteria.

`ssget` [str] [pts] [alist]	Builds and returns a selection set in a manner specified by the initial mode string argument or by the standard AutoCAD Select objects: mechanism.

`ssget` can be called in myriad ways to create a selection set in a manner that best suits a particular need.

In a new drawing, re-create the entities we drew at the beginning of this chapter and assign their entity names to `e1`, `e2`, and `e3` as we did before:

```
Command: (command "line" '(1 5) '(3 2) '(4 6) ""
                  "circle" '(8 2) 1.0)
nil
```

If called with no arguments, `ssget` uses AutoCAD's general Select objects: mechanism. The user then selects entities on the screen in the same manner as she does for, say, a COPY command.

```
Command: (setq ss1 (ssget))          ;"Select objects:" prompt
Select objects: c
First corner: Other corner: (draw a window across both lines) 2 found
Select objects:
<Selection set: 1>

Command: select
Select objects: !ss1 2 found         ;Both lines highlighted
```

We can use standard AutoCAD options such as Window and Crossing, and, where appropriate, the object snaps as well. Whatever entities we select are added to the selection set. The AutoCAD SELECT command provides a quick way to determine which entities a selection set contains.

If called with a single point, `ssget` returns the entity name of the entity that passes through that point.

```
Command: (setq ss2 (ssget '(3.5 4)))
<Selection set: 2>
```

If more than one entity passes through the supplied point, the results are indeterminant.

`ssget` can be called with one of several *modes*. A mode specification is one of the following strings, which can be specified in uppercase or lowercase:

`"L"`	"LAST"
`"P"`	"PREVIOUS"
`"W"`	"WINDOW"
`"WP"`	"WINDOW POLYGON"
`"C"`	"CROSSING"
`"CP"`	"CROSSING POLYGON"
`"F"`	"FENCE"
`"I"`	"IMPLIED"
`"X"`	"ALL"

The `"L"` mode builds a selection set containing only the last entity drawn, that is, `entlast`:

```
Command: (setq ss3 (ssget "L"))
<Selection set: 3>

Command: select
Select objects: !ss3 1 found          ;Circle highlighted
```

The `"P"` mode is identical to AutoCAD's Previous mode. It builds a selection set containing the same entity names that were in the previously created selection set.

```
Command: (setq ss4 (ssget "P"))       ;Most recently selected entities
<Selection set: 4>

Command: select
Select objects: !ss4 1 found          ;Circle highlighted
```

We would use this mode when we've selected several entities for an AutoCAD command, then realize that we want to perform more operations on those entities. The `"P"` option saves us the task of reselecting them.

The `"W"` mode takes the coordinates of the corner points of a window and builds a selection set of all entities that lie completely within that window.

```
Command: (setq ss5 (ssget "W" '(0 0) '(3.5 6.5)))    ;Entities inside of window
<Selection set: 5>

Command: select
Select objects: !ss5 1 found                          ;Only the first line is highlighted
```

The "C" mode is analogous to the "W" mode except that it finds all entities within a crossing window:

```
Command: (setq ss6 (ssget "C" '(0 0) '(3.5 6.5)))    ;Entities in crossing window
<Selection set: 6>

Command: select
Select objects: !ss6 2 found                        ;Both lines lie within crossing window
```

We use the "W" or "C" mode when we know the exact coordinates of a window or a crossing. If we want to allow the user to specify the coordinates, we run ssget without a mode string. Then the Select objects: prompt appears on the screen and the user can select entities in any manner that he chooses.

The "WP" and "CP" modes work similarly to "W" and "C" except that they find all entities within a polygon instead of a rectangular window. To demonstrate this we'll create a list of points and run ssget with both options:

```
Command: (setq p '((1 1) (4 8) (9 2)))
((1 1) (4 8) (9 2))

Command: select
Select objects: (ssget "wp" p))
1 found                                ;Second line is in window polygon

Select objects: (ssget "cp" p))
3 found (1 duplicate)                  ;All 3 entities are crossing polygon
```

Were a line to connect all of these points in a closed triangle, as shown in Figure 11-1, the polygon would completely window only the second line; thus the first selection set contains only one entity name. On the other hand, the triangle crosses or contains all three entities, so the "CP" option selects them all.

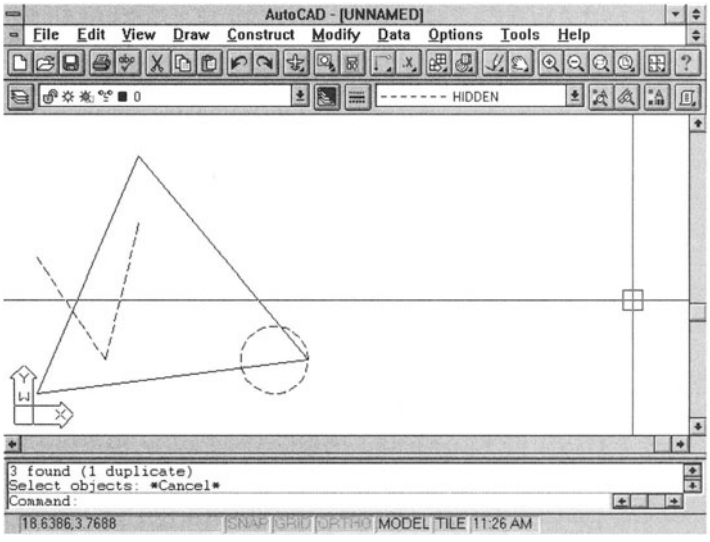


Figure 11-1

The `"F"` mode also takes a list of points that connect imaginary lines. It builds a selection set of all entities that the line would pass through.

```
Command: (setq ss9 (ssget "f" p))    ;Entities intersecting fence
<Selection set: 9>

Command: select
Select objects: !ss9 2 found         ;2 entities are fenced
```

Since the fence passes through the first line and the circle, `ss9` contains two entity names.

The `"I"` mode builds an implied selection set of all entities that have already been selected by the user. In order for this option to work, the PICKFIRST system variable must be set to 1. After verifying that it is set to 1 on your machine, click on one or more entities and enter the following function:

```
Command: (setq ss10 (ssget "i"))     ;Entities in implied selection set
<Selection set: 10>
```

When we examine `ss10`, we see that it contains the names of the entities that we selected prior to running `ssget`. If PICKFIRST is set to 0, `ssget` returns `nil` instead of creating the implied selection set.

The `"X"` mode builds a selection set of *every* entity in the drawing, including those on frozen and locked layers, in both model and paper space. Its use is shown by the `c:zapall` command, which erases all entities that are not on locked layers. We run it as follows:

```
Command: zapall
```

If we call `c:zapall` accidentally, we can run UNDO to bring back our erased entities. Here's its code:

```
(defun c:zapall (/ ss)
  (setq ss (ssget "x"))              ;SS of all entities
  (command "erase" ss "")            ;Erase them
  (redraw)                           ;Clean up the screen
  (prin1))
```

`c:zapall` creates a selection set of every entity in our drawing, then hands that selection set to AutoCAD's ERASE command. This command erases entities on frozen layers, something that ERASE ALL doesn't do. However, if we're in paper space, it won't delete entities in model space, and vice versa. To do that we would have to individually `entdel` each entity in the database.

Notes

- We usually issue `ssget` within a `setq` because we intend to perform operations on the selection set we're creating.
- We can open a maximum of 128 selection sets at a given time. If we try to create another one, `ssget` or `ssadd` returns `nil`.
- `ssget` and `ssadd` work with main entities only; a selection set can't contain subentities. This is why the `c:att-ed` command we studied in the last section couldn't use `ssget` to build the selection set of attributes.

11.2.1 Using Filters with `ssget`

The last optional argument to `ssget` is a filter list. A *filter list* is a list of dotted pairs that specifies the criteria for building a selection set. This association list looks quite similar to the entity data itself. It contains as many dotted pairs as are necessary to completely specify the entities that we want present in the new selection set.

For example, we might build a new selection set consisting of all of the circles in our drawing:

```
Command: (ssget "X" '((0 . "CIRCLE")))
<Selection set: 11>
```

We can get more specific. Let's construct a selection set of all of the red circles on Layer CUTOUTS:

```
Command: (ssget "X" '((0 . "CIRCLE") (8 . "cutouts") (62 . 1)))
<Selection set: 12>
```

8 is the group code for layer, 62 is the group code for color, and 1 is the color code for red. If there are no matches, `ssget` returns `nil`.

A given filter list *cannot* contain multiple entries with the same group code:

```
Command: (ssget "X" '((0 . "LINE") (0 . "CIRCLE")))
nil
```

This will *not* create a selection set of all of the lines and all of the circles in our drawing. We can obviate this restriction through the use of wildcards (defined in Section 9.1.1), as follows:

```
Command: (ssget "x" '((0 . "LINE,CIRCLE")))
<Selection set: 13>
```

Wildcards enable us to create a selection set in a variety of ways. We can select all objects on every layer whose name begins with `"XY"`:

```
Command: (ssget "x" '((8 . "XY*")))
<Selection set: 14>
```

We can build a selection set of all objects on Layer A and on every layer whose name begins with `"XY"`:

```
Command: (ssget "x" '((8 . "A,XY*")))
<Selection set: 15>
```

We can select all objects on Layers 1 to 4:

```
Command: (ssget "x" '((8 . "[1-4]")))
<Selection set: 16>
```

We can create a selection set of every entity that is *not* on Layer 0 or 1:

```
Command: (ssget "x" '((8 . "~[0-1]")))
<Selection set: 17>
```

Advanced Note: We must be careful when using filter lists to find anonymous blocks because their names begin with `*`, which is a wildcard character. We put a backquote character in front of the asterisk in the filter list to indicate that that the asterisk should be taken literally. For example, the following code builds a selection set containing anonymous Block *U0:

```
Command: (ssget "x" '((2 . "`*u0")))
<Selection set: 17>
```

♦ ♦ ♦

The use of filter lists is not limited to `ssget`'s `"X"` mode. As we saw earlier, if we provide no arguments to `ssget`, it uses the standard Select objects: mechanism. But if we give it a filter list, it adds only the appropriate entities to the selection set. In the following `ssget` call, regardless of what entities the user selects, only circles on Layer 1 are added to the selection set. All others are filtered out.

```
Command: (ssget '((0 . "circle") (8 . "1")))
<Selection set: 18>
```

The following function builds a selection set of the entities most recently selected, but only those that are circles:

```
Command: (ssget "p" '((0 . "circle")))
<Selection set: 19>
```

Example

`c:sclblk` asks the user to select one or more block inserts and enter a new scale factor. It then scales the selected block inserts up or down based on the user's input. (`c:sclblk` is based on a program by Tony Palmisano.) We run it as follows:

```
Command: sclblk
Select objects: (select one or more blocks)
Scale Factor (1 equals no change): .75
```

The selected blocks are now scaled to 3/4 their original size. Here's the code that accomplishes this:

```
  (defun c:sclblk (/ ss sf inc blk insp)
1   (initget 7)
    (setq ss  (ssget '((0 . "insert")))              ;SS of chosen inserts
          sf  (getreal "\nScale Factor (1 equals no change): ")
          inc -1)
```

```
2   (repeat (sslength ss)                          ;Get each block in SS
      (setq blk  (ssname ss (setq inc (1+ inc)))   ;Block's ename
            insp (cdr (assoc 10 (entget blk))))    ;Block's insert point
      (command "scale" blk "" insp sf))            ;Scale block
    (princ))
```

1. The `initget` applies to the following `getreal` function and ensures that a positive scale factor is assigned to `sf`.

   `ss` is bound to a selection set of all selected entities that are of type "insert." In other words, this `ssget` uses the standard AutoCAD Select objects: prompt, but it filters out everything that isn't an insert.

2. A loop is repeated once for each insert in the selection set. The entity name of the block is accessed by `ssname` and assigned to the symbol `blk`, and its insert point (group code 10) becomes the value of `insp`. The AutoCAD SCALE command then scales that block to the size requested by the user.

Advanced Note: Extended Entity Filtering

We can filter for all main entity types except entity names, handles, and extended entity group codes. `ssget` *can* locate, say, all lines having extended entity data with an the application ID of `"rh-rev"`:

```
Command: (setq ss5 (ssget "x" '((0 . "line") (-3 ("rh-rev")))))
<Selection set: 1>
```

Wildcards can also be used. The following function builds a selection set of all circles in the previous selection set having an Xdata application ID *beginning with* the string `"rh"` (this example does *not* follow from the last one):

```
Command: (setq ss6 (ssget "p" '((0 . "circle") (-3 ("rh*")))))
<Selection set: 6>
```

11.2.2 Advanced Topic: Relational Tests and Logical Operators

The filter lists we have built thus far use the = function for comparison. For example, we can build a selection set of all circles whose radius *equals* 2.0. With many of the filters we can use *relational tests,* much as we do with numeric comparisons. Filter lists accept the following relational operators:

Operator	*Meaning*
"="	Equal
"/=", "!=", or "<>"	Not Equal
"<"	Less than
"<="	Less than or equal
">"	Greater than
">="	Greater than or equal
"&"	Bitwise AND (integer groups only)
"&="	Bitwise masked equals (integer groups only)
"*"	Matches anything—always true

Relational filters introduce a new group code, –4, which is used differently from the other group codes we have seen. The cdr of a –4 pair is a string specifying one or more relational tests that apply to the *following dotted pair* in the entity data list. For example, this function builds a selection set of all circles whose radii *don't* equal 2.0:

```
Command: (ssget "x" '((0 . "circle") (-4 . "/=") (40 . 2.0)))
<Selection set: 1>
```

"/=" means "not equal," and 40 is the group code for a circle's radius.

Next, we'll build a selection set of all lines from the previous selection set whose starting X coordinate is greater than or equal to 8 and Y coordinate is greater than or equal to 6 (this example does *not* follow from the previous one):

```
Command: (ssget "p" '((0 . "line") (-4 . ">=,>=") (10 8 6)))
<Selection set: 4>
```

Now we'll create a selection set of all lines whose starting X coordinate is greater than 8 and Z coordinate equals 1. Since we don't care about the value of the Y coordinate, we use the "*" wildcard in the middle position to match any value.

```
Command: (ssget "x" '((0 . "line") (-4 . ">,*,=") (10 8 0 1)))
<Selection set: 5>
```

The "&" and "&=" relational operators are used to test bits in integer groups by applying a mask to the integer. "&" succeeds if *any* bit in the mask matches the corresponding bit in the number. "&=" succeeds if *every* bit in the mask

matches the corresponding bit in the number. If a test succeeds, the tested entity is added to the selection set.

In Chapter 10 we learned that polyline headers have a 70 group code that is set to 1 if the polyline is closed. The 70 group code can actually have a number of different bit code settings based on the type of polyline. For example, a bit setting of 8 indicates that the polyline is 3D. The following `ssget` builds a selection set containing the entity names of all closed 3D polylines in our drawing:

```
Command: (ssget "x" '((-4 . "&=") (70 . 9)))
<Selection set: 6>
```

A closed 3D polyline has bits 1 and 8 set, so we use `"&="` with a mask of 9 to test for them both.

♦ ♦ ♦

Relational tests perform binary comparisons: They only compare two operands. Furthermore, these operands must be numbers. Thus, we can't use relational tests to build a selection set of, say, all lines that are not on Layer 0, because `"0"` is a string:

```
Command: (ssget "x" '((0 . "line") (-4 . "/=") (8 . "0")))
nil
```

To accomplish this we must use logical operators. *Logical operators* provide an even more powerful means of specifying the entities we want added to a selection set by allowing us to compare groups of expressions. The available Boolean operators are shown in the following chart:

Starting Operator	*Number of Operands*	*Ending Operator*
`"<AND"`	One or more	`"AND>"`
`"<OR"`	One or more	`"OR>"`
`"<XOR"`	Two	`"XOR>"`
`"<NOT"`	One	`"NOT>"`

These Boolean operators are conceptually identical to the others we've seen throughout the book. They are explained in more detail in Section 14.2.3.

As with relational tests, the logical operators are best understood within the framework of a few examples. Load `examples.lsp` into a new drawing and run the `c:log-set` command:

```
Command: log-set
```

This command creates Layer 1 with the BYLAYER color red. It then draws four circles and two triangles in the following configuration:

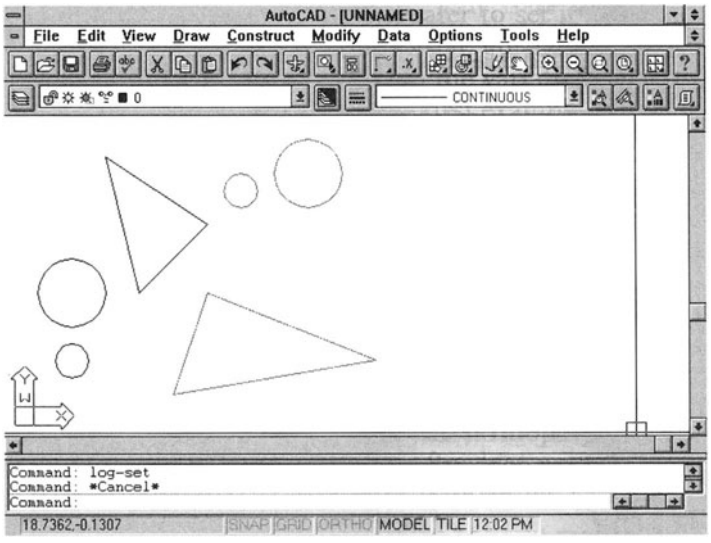


Figure 11-2

The circles and lines on the left are on Layer 0; the ones on the right are on Layer 1.

Example 1

We always begin a logical test with a –4 pair containing the name of the operator preceded by a *left* angle bracket. We end the test with the name of the same operator followed by a *right* angle bracket. In between we have one or more conditions (depending on the operator) to be operated on. Here's how we find all lines that are *not* on Layer 0:

```
Command: (setq ss1 (ssget "x"
                   '((0 . "line")       ;All lines
                     (-4 . "<not")      ; not
                       (8 . "0")        ; on Layer 0
                     (-4 . "not>"))))

<Selection set: 1>
```

```
Command: select
Select objects: !ss1                          ;Bottom triangle isn't on Layer 0
3 found
```

Example 2

We can nest logical operators to more precisely specify the entities we want included in the selection set. The `log1` function (in `examples.lsp`) builds a selection set of all circles with radius 0.5 and all lines on Layer 1:

```
Command: (setq ss2 (log1))
<Selection set: 2>

Command: select
Select objects: !ss2                          ;2 small circles & bottom triangle
5 found
```

Let's look at the code:

```
    (defun log1()
      (ssget "x"
3       '((-4 . "<or")
1         (-4 . "<and")
            (0 . "circle")                    ;All circles whose radius
            (40 . 0.5)                        ; equals 0.5
          (-4 . "and>")
2         (-4 . "<and")                       ;Plus
            (0 . "line")                      ; all lines
            (8 . "1")                         ; on Layer 1
          (-4 . "and>")
        (-4 . "or>"))))
```

1. `"<and"` indicates the beginning of a collection of tests to be ANDed together, and the `"and>"` three lines below designates the end. If an entity is a circle *and* it has a radius of 0.5, it is added to the selection set.

2. In the second `"<and"` group, if an entity is a line *and* it is on Layer 1, then it is added to the selection set.

3. The two `"<and"` groups are nested in an `"<or"` group, which specifies that the entity need only satisfy one of the `"<and"` groups to be included in the selection set. In other words, this function builds a selection set of all entities that are either circles with a radius of 0.5 *or* lines on Layer 1. Three lines and two circles satisfy this criteria.

Example 3

The following function builds a selection set of all circles whose radius does not equal 0.5, plus all circles whose center points have X and Y coordinates greater than 6. It excludes those circles that fulfill both requirements.

```
Command: (setq ss3 (log2))
<Selection set: 3>

Command: select
Select objects: !ss3                    ;Large circle on Layer 0 &
2 found                                 ; small circle on Layer 1
```

Here's the code:

```
  (defun log2 ()
    (setq ss2 (ssget "x"
3     '((-4 . "<xor")
1       (-4 . "<and")
          (0 . "circle")                ;all circles whose radius
          (-4 . "/=")                   ;does not = 0.5
          (40 . 0.5)
        (-4 . "and>")
2       (-4 . "<and")
          (0. "circle")                 ;all circles whose
          (-4 . ">, >")                 ;center points have
          (10 6 6)                      ;x > 6 and y > 6
        (-4 . "and>")
      (-4 . "xor>")))))
```

1. The first `"<and"` group finds entities that are circles that have a radius not equal to 0.5. The two large circles fit this criteria. Notice that the –4 group code is used for both the logical operator `"<and"` and the relational test `"/="`.

2. The second `"<and"` group finds entities that are circles with a center point whose X and Y coordinates are both greater than 6. The two circles on the upper right of the drawing are acceptable.

3. These two `"<and"` groups are then XORed. The `"<xor"` group specifies that any entity selected by one of the `"<and"` groups will be in the selection set *unless* that entity was selected by both tests. Since the large circle in the upper right has a radius greater than 0.5 *and* a starting X and Y coordinate greater than 6,6, it is excluded from the selection set. The resulting selection set, then, consists of the large circle on the left and the small circle on the right.

11.3 Nongraphical Entities

All entities we have seen to this point are graphical in nature; they can be drawn on the screen. There is a collection of entities that can't be drawn . . . they merely contain information that describes our drawing environment. There are two kinds of nongraphical entities, symbol tables and dictionaries.

11.3.1 Symbol Tables

Symbol tables are nongraphical entities that are used to store most of the nongraphical and some of the graphical entity information for our drawing. The following symbol tables exist and are accessible in AutoLISP:

APPID	Application IDs
BLOCK	Block definitions
DIMSTYLE	Dimension styles
LAYER	Layer information
LTYPE	Linetypes
STYLE	Text styles
UCS	User coordinate systems
VIEW	Named views
VPORT	Viewports

The information about each table entry is kept in an association list that is quite similar to the entity data lists, although the group codes for tables are different. To some extent we can manipulate table data in much the same way that we manipulate graphical entity data.

`tblnext str [t]`	Takes a table name and returns the next entry in that table.

The first time it is called, `tblnext` returns the first entry in the specified table. Each time it is called thereafter for that table it returns the next succeeding entry. When it reaches the end of a table, `tblnext` returns `nil`.

To see how tables work, begin a new drawing and create a layer named 1. Assign to it the color blue and the linetype hidden. There are now two entries in the Layer table:

```
Command: (tblnext "layer")
((0 . "LAYER") (2 . "0") (70 . 0) (62 . 7) (6. "CONTINUOUS"))

Command: (tblnext "layer")
((0 . "LAYER") (2 . "1") (70 . 64) (62 . 5) (6. "HIDDEN"))

Command: (tblnext "layer")
nil
```

The group codes have the following meanings for layers:

0	Table name
2	Layer name
70	Flag bits
62	Color
6	Linetype

The color and linetype are our "BYLAYER" information.

The flag bits for a layer provide such information as whether the layer is frozen or locked. If we've bound the symbol `flag` to the flag bits for Layer 1, we can test whether the layer is frozen as follows:

```
Command: (logand flag 1)
0
```

Bit 1 indicates whether the layer is frozen. If it is, `logand` returns 1; otherwise it returns 0.

Each table type has its own set of codes. The group codes for each table are listed in the *AutoLISP Reference.* Release 12: Appendix B; Release 13: Chapter 16.

A Block table entry might appear as follows:

```
Command: (tblnext "block")
((0 . "BLOCK") (2 . "PENT") (70 . 66) (10 0.0 0.0 0.0) (-2 . <Entity name: 88440548>))
```

Block table entries have 0, 2, and 70 pairs that keep track of the table name, block name, and flag bits, respectively. They also have a 10 field, which gives the block's base point, and a –2 field, which gives the entity name of the first entity in the block.

We can extract the –2 field and supply the entity name to `entget` and `entnext` to see the entity data and locate the next entity name, respectively. Furthermore, we can change the entity data and `entmod` it, thereby altering an

entity in the block definition. We can't add the entity name to a selection set, however.

See Section 10.3 to learn how to entmake blocks.

We can ask tblnext to rewind and retrieve the first entry for a table by supplying the symbol t as an optional second argument:

```
Command: (tblnext "layer" t)
((0 . "LAYER") (2 . "0") (70 . 0) (62 . 7) (6 . "CONTINUOUS"))
```

The next time we call tblnext for the Layer table, it will return the second entry.

tblsearch str str [t]	Takes a table name and the name of a table entry and returns that entry.

We use tblsearch when we want to see a specific entry or when we just want to test whether there is an entry with that name. We would retrieve the record for Layer 1 as follows:

```
Command: (tblsearch "layer" "1")
((0 . "LAYER") (2 . "1") (70 . 64) (62 . 5) (6 . "HIDDEN"))
```

tblnext normally works independently of tblsearch. It sequences through the table entries in order, regardless of any retrievals performed by tblsearch. However, if we supply t as an optional third argument to tblsearch, the next time tblnext is called it returns the entry following the one retrieved by tblsearch:

```
Command: (tblsearch "layer" "1" t)
((0 . "LAYER") (2 . "1") (70 . 64) (62 . 5) (6 . "HIDDEN"))
```

In this example tblsearch tells tblnext to return the entry following this one the next time we use it to fetch a record from the Layer table. This option is especially suitable for working with viewports, as the following example shows.

Example 1

The Vport table contains an entry for each named viewpoint. If we divide the drawing screen into four viewports and save that configuration, four table en-

tries will be created. Each entry contains the coordinates of its lower left (group code 10) and upper right (group code 11) corners (the same data returned by the vports function), among other information.

Let's split the screen into two vertical viewports and call this configuration "V2". Two table entries are created:

```
Command: (tblnext "vport")
((0 . "VPORT") (2 . "V2") (10 0.5 0.0) (11 1.0 1.0) . . .)

Command: (tblnext "vport")
((0 . "VPORT") (2 . "V2") (10 0.0 0.0) (11 0.5 1.0) . . .)
```

Consider that we might have saved three-viewport and four-viewport configurations as well. If we want to access the viewport entries for "V2", we can find the first one with tblsearch, then sequence through the others with tblnext. Here's a code fragment that we might use to accomplish this:

```
(setq view (tblsearch "VPORT" "V2" t))        ;1st entry for V2
(while (and view                              ;Any entries left?
         (eq (cdr (assoc 2 view)) "V2"))      ;Is it still V2?
             .
             .                                ;Do meaningful work here
             .
  (setq view (tblnext "vport"))               ;Get next entry
)
```

This function finds the first entry for "V2". If there are none, view is set to nil and the and fails. Otherwise, the code processes the first "V2" entry, then gets the next table entry. tblnext is initially positioned to the correct entry by the t argument to the tblsearch function.

Example 2

The following useful command, c:new-style, prompts for the name of a text style. If the style is found in the Style table, it is made the current style. Otherwise the command creates a new style based on SIMPLEX, with a height and width supplied by the user. Other fields take the default. (c:new-style is based on a program by Tracy Weaver.)

Here's an example of how we'd run the command, followed by the code itself:

```
Command: new-style
Enter new style: tall
Creating style TALL
height: 4
width: .3
TALL is now the current style.
```

```
  (defun c:new-style (/ style height width)
1   (setq style (getstring "\nEnter new style: "))
2   (if (tblsearch "STYLE" style)                          ;Does this style exist?
        (setvar "textstyle" style)                         ;Yes, use it
3       (progn                                             ;No, create it
            (prompt (strcat "\nCreating style" (strcase style)))
            (initget 7)
            (setq height (getdist "\nHeight: "))
            (initget 7)
            (setq width (getdist "\nWidth: "))
4           (command "STYLE" style "SIMPLEX" height width "" "" "" "")))
    (prompt (strcat "\n" (strcase style) " is now the current style."))
    (princ))
```

1. `c:new-style` requests the name of a style and assigns it to the variable `style`. It uses `getstring`, not `getstring t`, because the style name cannot contain spaces.

2. It searches the Style table for the entered style and, if it's found, sets the STYLE system variable to reflect the user's selection.

3. If the style is not in the table, then it must be created. An `initget` value of 7 prevents the user from typing <enter>, zero, or a negative value for the height. `getdist` is used instead of `getreal` so that the user can enter the height by clicking on the screen, thereby effectively saying, "I want the height of the text to be the same as this measurement." These steps are repeated to obtain the width.

4. The STYLE command is called with the information that the user supplied. The null strings indicate that the defaults are to be used for such questions as "Backwards?" and "Vertical?".

Example 3

This last example wriggles a little deeper into the entity database to extract and return the text value in an associative dimension. Although it goes a little beyond simply accessing a table, it's an invaluable program if you need to extract dimension information.

Suppose we have the following associative dimensions in our drawing:

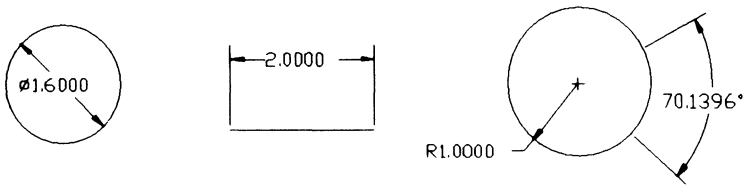


Figure 11-3

We can learn the value of any of these measurements as follows:

```
Command: dimsize
Select dimension entity: (click on angle measurement)
70.1396
```

`c:dimsize` asks the user to select a dimension, then returns the real number value of that dimension. A dimension is actually a block, and our code must cycle through all of its component entities until it locates the text.

The dimension text itself is a string that can be converted to a real number with `read`. However, this string might have extraneous characters that must be eliminated in order for the conversion to work. Furthermore, the text is a text entity in Release 12 and an mtext entity in Release 13. A conversion function for each release is provided below.

The Release 12 version of this function in the `examples.lsp` file is called `c:dimsize12`. It is identical to the following Release 13 function, except that on the line numbered 2 it tests for `"TEXT"` instead of `"MTEXT"`, and on the line numbered 3 it calls the `stor12` function instead of `stor13`.

```
  (defun c:dimsize (/ e dimnam dimtab de size)
1   (setq e (car (entsel "\nSelect dimension entity: ")))
    (while (or (null e)                        ;Nothing selected or invalid dim
               (not (eq (cdr (assoc 0 (entget e))) "DIMENSION")))
      (setq e (car (entsel "\nNot a dimension entity. Select again: "))))
    (setq dimnam (cdr (assoc 2 (entget e))))   ;Dimension name, e.g. "*D2"
    (setq dimtab (tblsearch "block" dimnam))   ;Dimension data
    (setq de (cdr (assoc -2 dimtab)))          ;Dimension component ename
2   (while (not (eq (cdr (assoc 0 (entget de))) "MTEXT"))   ;Rel12: Search for TEXT
      (setq de (entnext de)))                  ;Scan until text encountered
    (setq size (cdr (assoc 1 (entget de))))    ;Get size text
3   (setq size (stor13 size))                  ;Convert to real number. R12: call stor12
    (terpri)                                   ;Send result to a newline
    size)                                      ;Return Size
```

1. `c:dimsize` begins by asking the user to select a dimension entity. If the user chooses an entity other than a dimension (or misclicks and selects nothing, in which case `entget` returns `nil`), the `while` loop reprompts.

   When a dimension entity is finally selected, the 2 field is found in the entity data. This is the dimension name, and it's used to locate the dimension entry in the Block table. The –2 field is extracted from this entry and assigned to the variable `de`. This is the name of the first entity in the block.

2. The command next sequences through all of the entities that constitute the dimension until it locates the text. When it finds the mtext entity, it extracts the value of the 1 group code, which is the text that specifies the dimension size.

3. The command finally has the string it needs, so it calls the `stor13` function to convert it to a real number and returns that number.

We'll now examine the `stor13` code. If you're running Release 12, read the explanation of `stor12` (which follows the Release 13 explanation) instead.

Release 13

```
   (defun stor13 (text)
1    (setq text (substr text 5))                          ;Get past initial garbage
2    (cond ((= (substr text 1 7) "\\U+2205")              ;1st chars = Diameter?
            (read (substr text 8)))                       ;  skip 1st 7 chars.
3          ((= (substr text 1 1) "R")                     ;1st char = R = Radius
            (read (substr text 2)))                       ;  skip 1st char.
4          ((= (substr text (strlen text)) (chr 176))     ;Last char = degrees?
            (read (substr text 1 (- (strlen text) 1))))   ;  skip last char.
5          (t (read text))))                              ;No stripping, just convert
```

`stor13` (***S**tring **TO R**eal*) uses the `read` function to convert a dimension text string to a real number. But it must perform some preprocessing:

1. The mtext string for all dimension text begins with `"\A1;"`. These four characters must be stripped off the front of the string. For this reason `cond` is passed a substring of the original text string, beginning at character position 5.

2. If the string is a diameter measurement, then it begins with `"\U+2205"`. These seven characters, which are the UNICODE representation of the diameter symbol, must be stripped off. Thus `read` converts a substring starting at character position 8.

3. If the string is a radius measurement, then it begins with the letter `"R"`. In that case the first character only is bypassed for the conversion.

4. If the string is an angle measurement, then it *ends* with the degree sign, `"°"`. `stor13` checks for this by giving its ASCII value, 176, to the `chr` function. If there's a match, then the last character is bypassed for the conversion.

5. If no test succeeds, then the string is a linear measurement. In that case no characters need to be stripped; the string is simply converted to a number by `read`.

Note: The dimension text might be represented differently on different systems. For example, instead of using `(chr 176)` to specify the degrees sign, you might have to use its Unicode equivalent, `"\U+00B0"`.

Release 12

```
  (defun stor12 (text)
1   (cond ((= (substr text 1 1) "%")                    ;1st char = %?
           (read (substr text 4)))                      ;  skip 1st 3 chars.
2         ((= (substr text 1 1) "R")                    ;1st char = R?
           (read (substr text 2)))                      ;  skip 1st char.
3         ((= (substr text (strlen text)) "d")          ;last char = d?
           (read (substr text 1 (- (strlen text) 3))))  ;  skip last 3 chars.
4         (t (read text))))                             ;No stripping, just convert
```

`stor12` (***S***tring ***TO R***eal) uses the `read` function to convert a dimension text string to a real number. But it must perform some preprocessing:

1. If the string is a diameter measurement then it begins with `"%%c"`. Thus, if the first character is `%`, then the first three characters are bypassed for the conversion.

2. If the string is a radius measurement, then it begins with the letter `"R"`. In that case the first character only is bypassed for the conversion.

3. If the string is an angle measurement, then it *ends* with `"%%d"`. Therefore, if the last character is `"d"`, the last three characters are bypassed for the conversion.

4. If no test succeeds, then the string is a linear measurement. In that case no characters need to be stripped; the string is simply converted to a number by `read`.

♦ ♦ ♦

The remainder of this section discusses features that were introduced in Release 13.

`tblobjname str str`	Takes a table name and the name of an entry and returns the entity name for that entry.

Each symbol table entry has its own entity name through which we can access the entry in a manner similar to the graphical entities. For example, we can `entget` the entity data for an entry, alter it, and `entmod` the entry to reflect the change.

In the preceding Example 2, we created a style named TALL whose height is 4.0. If we really meant it to be 0.4, we can change it as follows:

```
Command: (setq s (tblobjname "style" "tall"))
<Entity name: 88450778>

Command: (setq sd (entget s))
((-1 . <Entity name: 88450778>) (0 . "STYLE") (5 . "6F") (100 . "AcDbSymbolTableRecord")
 (100 . "AcDbTextStyleTableRecord") (2 . "TALL") (40 . 4.0) (41 . 0.3) (50 . 0.0)
 (71 . 0) (42 . 4.0) (3 . "SIMPLEX") (4 . ""))

Command: (entmod (subst '(40 . 0.4) (assoc 40 sd) sd))
((-1 . <Entity name: 88450778>) (0 . "STYLE") (5 . "6F") (100 . "AcDbSymbolTableRecord")
 (100 . "AcDbTextStyleTableRecord") (2 . "TALL") (40 . 0.4) (41 . 0.3) (50 . 0.0)
 (71 . 0) (42 . 4.0) (3 . "SIMPLEX") (4 . ""))
```

We use `tblobjname` to learn the entry's entity name and `entget` to access its data. Thereafter, we assign it a new text height (40 group code) and `entmod` the entity's data. The next time we call the TEXT command, the text height will be 0.4.

`snvalid str`	Returns `t` if the argument is a valid symbol table name and `nil` otherwise.

A symbol table name can contain alphanumeric characters as well as the special characters hyphen `(-)`, underscore `(_)`, and dollar sign `($)`.

We can use `entmake` in conjunction with `tblobjname` to create a new symbol table entry. If we request a name from the user, `snvalid` can be used to verify that the symbol name is valid. Note that it does not test whether that name is already being used; we use `tblsearch` for that. We might run `snvalid` as follows:

```
(while (not (snvalid (setq name (getstring "Enter new layer name: "))))
  (princ "\nInvalid name. "))
```

This loop reprompts until `snvalid` approves the user's input.

Now let's see how we'd actually make a new layer. We can go about it in two ways. Having earlier created Layer 1, we can make a layer named NEW by modifying the Layer 1 data. First, we'll get the entry's entity name and entity data:

```
Command: (setq l (tblobjname "layer" "1"))
<Entity name: 88450568>

Command: (setq ld (entget l))
((-1 . <Entity name: 88450568>) (0 . "LAYER") (5 . "2D") (100 . "AcDbSymbolTableRecord")
 (100 . "AcDbLayerTableRecord") (2 . "1") (70 . 64) (62 . 5) (6 . "HIDDEN"))
```

Next, substitute the new data for the old and make the entity:

```
Command: (entmake (subst '(2 . "new") '(2 . "1") ld))
((-1 . <Entity name: 88450568>) (0 . "LAYER") (5 . "2D") (100 . "AcDbSymbolTableRecord")
 (100 . "AcDbLayerTableRecord") (2. "new") (70 . 64) (62 . 5) (6 . "HIDDEN"))
```

We now have a table entry for Layer NEW. Another way to do this is to build the new entry from scratch:

```
Command: (entmake '((0 . "LAYER") (100 . "AcDbSymbolTableRecord")
         (100 . "AcDbLayerTableRecord") (2 . "next") (70 . 64) (62 . 1)
         (6 . "CONTINUOUS")))
((0 . "LAYER") (100 . "AcDbSymbolTableRecord") (100 . "AcDbLayerTableRecord")
  (2 . "next") (70 . 64) (62 . 1) (6 . "CONTINUOUS"))
```

All of the preceding group codes are required, except for the 62 and 6 fields. Furthermore, the 100 strings must maintain the uppercase and lowercase shown here.

Advanced Note: Extended Entity Data

We can add extended entity data to each symbol table entry. For example, let's add Xdata to our Layer 1 entry. First we'll register an application, then we'll create new Xdata containing the string `"Try this"`:

```
Command: (regapp "RH-TEST")
"RH-TEST"

Command: (setq app '(-3 ("RH-TEST" (1000 . "Try this"))))
(-3 ("RH-TEST" (1000 . "Try this")))
```

Next we'll append the Xdata to Layer 1's entity data (which we retrieved earlier), and modify the database:

```
Command: (entmod (append ld (list app)))
((-1 . <Entity name: 88450568>) (0 . "LAYER") (5 . "2D") (100 . "AcDbSymbolTableRecord")
 (100 . "AcDbLayerTableRecord") (2 . "1") (70 . 64) (62 . 5) (6 . "HIDDEN")
 (-3 ("RH-TEST" (1000 . "Try this"))))
```

Now let's get the entity data once again to verify that the Xdata is there:

```
Command: (entget l '("RH-TEST"))
((-1 . <Entity name: 88450568>) (0 . "LAYER") (5 . "2D") (100 . "AcDbSymbolTableRecord")
 (100 . "AcDbLayerTableRecord") (2 . "1") (70 . 64) (62 . 5) (6 . "HIDDEN")
 (-3 ("RH-TEST" (1000 . "Try this"))))
```

11.3.2 Advanced Topic: Dictionaries

There is another nongraphical entity type called a *dictionary,* which is a mechanism for storing information. Starting with Release 13, all new nongraphical entity types (which are now known as *objects*) will be stored in dictionaries rather than having their data represented in symbol tables.

`namedobjdict`	Returns the entity name of the current drawing's named object dictionary.

`namedobjdict` takes no arguments and returns the entity name of the current drawing's *named object dictionary,* which is the root dictionary for the current drawing. This entity name can be used to access other dictionaries in a treelike fashion.

```
Command: (setq d (namedobjdict))
<Entity name: 88450460>
```

Dictionaries provide a means for storing information about the multiline style and group objects, plus any new object types added to AutoCAD in the future. The main dictionary encompasses the ACAD_MLINESTYLE and ACAD_GROUP dictionaries, which contain all of the multiline style objects and groups, respectively, in the drawing. Each group in the ACAD_GROUP dictionary has its own dictionary containing the entities within that group.

`dictnext` ename {`t`}	Takes the entity name of a dictionary and returns the next entry in that dictionary.

```
Command: (setq d1 (dictnext d))
((-1 . <Entity name: 88450470> (0 . "DICTIONARY") (5 . "E") (102 . "{ACAD_REACTORS")
 (330 . <Entity name: 88450460>) (102 . "}") (100 . "AcDbDICTIONARY") (3 . "STANDARD")
 (350 . <Entity name: 88450498>))
```

This is the first entry in the named object dictionary, which is the ACAD_MLINESTYLE dictionary. Each time we run `dictnext`, it returns the succeeding dictionary entry:

```
Command: (setq d2 (dictnext d))
((-1 . <Entity name: 88450468> (0 . "DICTIONARY") (5 . "D") (102 . "{ACAD_REACTORS")
 (330 . <Entity name: 88450460>) (102 . "}") (100 . "AcDbDICTIONARY") (3 . "GGG")
 (350 . <Entity name: 88450718>) (3 . "HHH") (350 . <Entity name: 88450720>)
 (3 . "FFF") (350 . <Entity name: 88450728>))
```

This is the ACAD_GROUP entry, which contains records for groups named `"FFF"`, `"GGG"`, and `"HHH"`. `dictnext` returns `nil` when the end of the dictionary is reached. If at any time we supply `dictnext` with the optional argument of `t`, it rewinds and returns the first entry in the dictionary.

`dictnext` does for dictionaries what `tblnext` does for tables. Another function named `dictsearch` works analogously to `tblsearch`.

`dictsearch` ename sym {`t`}	Takes the entity name of a dictionary and the name of a dictionary entry and returns that entry.

Having bound `d2` to the ACAD_GROUP entry, we can get the ename for the ACAD_GROUP dictionary:

```
Command: (setq gn (cdr (assoc -1 d2)))
<Entity name: 88450468>
```

We can use this entity name to search for a group named `"HHH"`:

```
Command: (setq gd (dictsearch gn "hhh"))
((-1 . <Entity name: 88450720> (0 . "GROUP") (5 . "64") (102 . "{ACAD_REACTORS")
 (330 . <Entity name: 88450468>) (102 . "}") (100 . "AcDbGROUP") (300 . "") (70 . 0) (71 . 1)
 (340 . <Entity name: 88450668>) (340 . <Entity name: 88450700>)
 (340 . <Entity name: 88450660>) (340 . <Entity name: 88450708>))
```

The 340 group code indicates the entity name of an entity in the group. Once we have a dictionary entry for a group, we can access individual entities in the following manner:

```
Command: (setq e (cdr (assoc 340 gd)))
<Entity name: 88450668>

Command: (entget e)
((-1 . <Entity name: 88450668>) (0 . "CIRCLE") (5 . "4D") (100 . "AcDbEntity") (67 . 0)
 (8 . "0") (100 . "AcDbCircle") (10 2.0 4.0 0.0) (40 . 1.0) (210 0.0 0.0 1.0))
```

`entget` returns the entity data for the first circle in the `"HHH"` group.

`dictnext` normally sequences through a dictionary in order, regardless of any retrievals performed by `dictsearch`. However, as with `tblsearch`, if we supply `dictsearch` with an optional third argument of `t`, `dictnext` subsequently returns the entry following the one retrieved by `dictsearch`.

Notes

- Be aware that the *AutoLISP Reference* mistakenly gives `dictnext` the definition for `dictsearch`.
- The dictionaries used in AutoLISP are unrelated to the dictionaries used by AutoCAD's spell checker.
- As with symbol tables, extended entity data can be attached to dictionary entries.

11.4 Important Points to Remember

- The printed representation of a selection set tells us nothing about the contents of the selection set.

  ```
  Command: !ss3
  <Selection set: 7>
  ```

  Be careful not to confuse the decimal number in the printed representation (7) with the number used in the symbol we've assigned to the selection set (3).

- `ssget` can build a selection set in one of several ways:
  - If called with no arguments, `ssget` uses the standard AutoCAD Select objects: prompt.
  - If called with a mode string, such as `"W"` or `"P"`, `ssget` builds a selection set according to that mode.
  - If called with a filter list, `ssget` only adds entities that satisfy the supplied filters. The filter list can be used with any of the modes or with no mode string at all.
- `ssget "X"` mode is almost always used with filters. Without them, `ssget "X"` mode builds a selection set of every entity in the drawing, including those on frozen and locked layers (but we can't edit entities that are on locked layers).

11.5 Labs

Selection Sets

1. Load the `examples.lsp` file into a fresh drawing and run the `c:ss-set` command to draw some entities on your screen. Then perform the following tasks at the Command: prompt and answer the questions below. Remember that we can supply a selection set to AutoCAD's SELECT command to learn which entity names it contains.

   a) Build a selection set of all of the entities in the drawing and make it the value of `ss1`. How many entities are in this selection set?

   b) The black circles are the first two entities in the drawing. Name them `e1` and `e2` and remove them from `ss1`.

   c) Create a new selection set `(ss2)` containing only `e1`. Then add `e2` to that selection set.

   d) Build a selection set `(ss3)` containing the entity names of all of the circles in the drawing. What is the length of this selection set?

   e) Make a new selection set `(ss4)` consisting of all of the entities in `ss3` that are on Layer 1. Within a `command` function move all of the entities in this selection set 0.5 units to the right.

Steps f) and g) require that you've read the Advanced Topic on relational tests and logical operators.

f) Create a selection set (`ss5`) containing all text entities whose text height is less than 0.4. The group code for text height is 40.

g) Write a command that assigns to `ss6` a selection set containing the two instances of `"This is Text"` and all of the circles whose radius is greater than 0.5. This is involved enough that you probably want to write it using an editor.

h) Make a new selection set (`ss7`) containing the two instances of `"This is Text"`.

2. Augment the `c:find-circles` command we saw in Section 10.3 to request a selection set name from the user and add each circle to that selection set. It should also print each circle's layer along with its center point and radius. Note that this command requires the `otos` and `tab` functions that we wrote in Chapter 8 and 9 labs (they're in `examples.lsp`):

```
Command: f-c
Selection set name: ss1

     Center point          Radius     Layer

(9.3209 5.79506 0.0)     0.961436     "0"
(5.16674 4.36187 0.0)    0.958669     "0"
(3.89453 7.47749 0.0)    1.54286      "1"

Command: (sslength ss1)
3
```

3. Write a command that prompts the user to select objects, then moves them to the current layer. The name of the current layer is stored in the CLAYER system variable.

```
Command: cl
Select objects: (select entities)
```

The selected entities are moved to the current layer.

4. Write a function that takes a selection set and an entity type as arguments. Have it search the selection set and remove all entities of that type.

```
Command: (ss-sieve ss1 "circle")                ;Remove all circles
```

Notes

- If you prefer, have the function remove everything *except* the specified entity type.
- If you prefer, make this a command instead of a function and request the needed data from the user.
- You may remove entities from the existing selection set or create a new selection set minus the specified entities. In the latter case you should have the function return the selection set as a result so that it can be assigned to a symbol.
- There are at least two radically different approaches to this problem; see if you can discover them both.

5. Write a command to alter polyline widths. Have the user select the plines to change and enter the new width, then alter all of the polylines in the selection set to have the new width. Note that you must update both the 40 and 41 group code entries (try modifying only one and see what happens!).

   In Release 13 you must alter each vertex subentity, whereas in Release 12 you need only change the fields in the polyline header record. If you're working in Release 12, we suggest that you have your command alter the subentities anyway so that it is upward compatible.

6. In Section 11.2.1 we wrote the command `c:sclblk` to scale blocks. Modify this command so that it gives the user the choice of either selecting random blocks to scale, as it now does, or clicking on one block insert and having all inserts of that block scaled by the same factor. Note that the `y-np` function from the Chapter 8 labs is useful for this exercise.

```
Command: blkscale
Change all occurrences of the Block? (Yes/<No>): y
Select one of the Blocks: (select a block)
Scale Factor (1 equals no change): 1.5
```

   All inserts of the selected block are scaled 50 percent larger.

Nongraphical Entities

7. Write a command that returns a count of the number of layers in the drawing.

```
Command: lay-num
3
```

8. Write a function that takes the name of a table and prints on the screen all of the entries for that table.

```
Command: (tlist "layer")
((0 . "LAYER") ( 2 . "0") (70 . 0) (62 . 7) (6 . "CONTINUOUS"))
((0 . "LAYER") ( 2 . "1") (70 . 64) (62 . 3) (6 . "CONTINUOUS"))
```

9. Write a command that prompts the user to select an entity, then returns the color code for that entity, even if the entity's color is BYLAYER.

```
Command: col
Select object: (select entity on green layer)
3
```

10. Recall the `c:nl` command that we wrote in the Entities chapter, Lab Exercise 4. If the user specifies a layer that doesn't exist, AutoCAD will create a new (and perhaps unwanted) layer. Modify this code to prompt the user with the names of the existing layers.

```
Command: nl2
Select object: (click on an entity)
New layer for this entity <0 1 A>: 2
Invalid option keyword

New layer for this entity <0 1 A>: a
```

The selected entity is moved to Layer `A`.

11. Sometimes we want to temporarily thaw some layers, then return our layer configuration to its previous state. Write a command that creates a list of the names of all the frozen layers in the drawing and assigns that list to a global variable. Next, thaw several of the frozen layers. Then write a second command that returns the layer configuration to its original state. A value of 1 in a layer's flag bits (70 group code) indicates that the layer is frozen. Hint: Use `logand`.

12. Write a command that creates a new text style with `entmake`. It should request a style name, height, and width from the user, and modify the STANDARD text style (or build a text style entry from scratch). This command requires Release 13.

V

Becoming Better AutoLISP Programmers

Wouldn't you like to write efficient programs that run correctly the *first time?* There's more to being a good programmer than the ability to create a working piece of code.

Chapter 12 explores many causes of errors and how to rapidly recover from them. We'll see how the error messages themselves are occasionally misleading and explore situations in which incorrect code will give a wrong answer rather than an error message. We'll also learn how to enhance our programs by anticipating and preparing for erroneous input from our users.

Programming style is an important subject that is rarely considered. Let's write elegant programs that will make sense to someone else in our group or to ourselves when we pick them up six months later. In Chapter 13 we'll explore the correct ways of using variables and discuss issues involved with combining functions into a large program. We'll also learn how to write really robust code that can take a lot of pounding without breaking.

The new AutoCAD help facility allows us to provide help for the AutoCAD commands that we write. In this chapter we'll learn how to set up and use this facility.

12

Error Handling and Debugging

Topics Covered in This Chapter

- Frequently encountered AutoLISP error messages
- Error-handling subrs
- Effective debugging techniques

Goals for This Chapter

- Understand the causes of common error messages in order to minimize debugging time.
- Learn the contracts of functions that facilitate error reporting and debugging.
- Examine code with errors and study ways to approach debugging it.

Introduction

AutoLISP has a very thorough error-reporting mechanism, but its debugging facilities are comparatively weak. We can maximize our productivity by understanding the various error messages and what causes them, and by knowing the best ways to avoid errors in the first place.

Debugging code goes beyond just knowing what the error messages mean. The difficult bugs to find are those that don't yield error messages at all or appear only sporadically. Some errors can arise from the user entering the wrong data or pressing the <esc> key; our code should be robust enough to handle these circumstances.

In this chapter we'll first go over some common error messages and learn how to recover from them very quickly. Next, we'll see some subrs that enable our programs to report errors or aid in the debugging process. Finally, we'll look at several examples of erroneous code and study some methods for locating and correcting the errors.

12.1 Frequently Encountered AutoLISP Error Messages

AutoLISP's error messages are quite helpful in pinpointing the bugs in our code . . . if we know how to interpret them. The most important fact to remember about error messages is that the *first* line of code following any error message always displays the *exact* place where the error occurred.

An error in a large program might print out many lines of code, so many in fact that we might not see the line where the error occurred . . . it scrolls right off of the screen. It can happen with small programs too, if there are numerous function calls. AutoCAD for Windows allows us to look back at the command history, but the DOS version doesn't afford that luxury. DOS users can turn on Printer Echo by entering `Ctrl-Q`, run the offending function again, then toggle Printer Echo off by entering `Ctrl-Q` once more. We then have a hard copy listing that shows the entire error output.

Errors can crop up in a variety of places. Sometimes we get error messages when we load a function. Several errors are reported by `eval`. Others are returned by the functions themselves. We'll look at the causes of many possible errors and what each message means.

If we load a file or run a function from the graphics screen, there may not be enough room in the little text window for AutoLISP to print an error message. If an error occurs, `*Cancel*` is printed following the error message. When we

see `*Cancel*` we should flip to the text screen to read the error message. If we're using AutoCAD for Windows, we can expand the graphic screen's text window when we plan to work with AutoLISP for a while.

We've discussed errors in various parts of this book. You can look up one or more error messages under "errors" in the index for further discussion of their causes.

12.1.1 Errors Encountered When Loading Files

If an obvious error exists in our `.lsp` file, AutoLISP issues an error message when we attempt to load it. We must correct the error and reload the file before attempting to run the code.

LOAD failed

One error message we get when loading a file is the "LOAD failed" error:

```
Command: (load "text")
error: LOAD failed.
```

This error arises when AutoLISP can't find the file we specified. Here are several reasons that this may happen:

- We misspelled the filename. Maybe the real name of the file we just tried to load is `test.lsp`.
- When we edit a file, we must specify the complete filename (e.g., `ahed test.lsp`), but when we load it we usually leave off the `.lsp` extension. It's easy to become confused and accidentally edit a new file without the extension (e.g., `ahed test`). Then, when we try to load it, there is no file named `test.lsp` and the `load` fails.
- The `load` function searches the AutoCAD Library Path for files. The first directory on the Library Path is the current directory, which is the directory from which we loaded AutoCAD. If we change directories once AutoCAD is loaded, `load` may not find our files.

  If we have a program directory that we use most or all of the time (generally a good idea), then this problem can easily be overcome. In Windows we can set the Working Directory in the File Properties menu for AutoCAD to our program directory.

In DOS (and Windows too) we can set the `acad` system environment variable (described in Section 8.4) to one or more directories. When a drawing is loaded, the `acad` variable is read by AutoCAD and the specified directories are placed on the AutoCAD Library Path.

The `acad` system environment variable is set in the `acadr13.bat` file for DOS and in the `acadr13\win\acad.ini` file for Windows. If you can't find the proper file, consult your system manager to see if it's been moved. Once you've located the file you can edit it to add your program directory. Thereafter, when you enter AutoCAD, `load` will search that directory for your AutoLISP files.

In our programs we might ask the user to supply the name of a file to load. If the user supplies an erroneous filename, we don't want our program to crash. `load` takes an optional second argument that is evaluated if the file does not exist. The following code reprompts the user until a valid filename is entered:

```
(setq fname (getstring "\nEnter name of file to load: "))
(while (null (load fname nil))
  (setq fname (getstring "\nFile doesn't exist. Enter new filename: ")))
```

In this example, if `load` fails it returns `nil`, `null` returns `t`, and the `while` repeats. An alternative solution is to use `getfiled`, which prompts the user to choose from a list of existing files.

malformed list

We get a "malformed list" error when we load a file that is missing a close parenthesis. Since our `defuns` are actually lists, this is AutoLISP's way of telling us that we began a list that we never finished.

The missing parenthesis might belong anywhere in the file; it doesn't necessarily go at the end. Because missing parentheses are sometimes hard to locate, it is highly recommended that we enter our code with AHED or another text editor that balances parentheses.

If we forget a close parenthesis when we type a function at the Command: prompt, AutoCAD issues a `1>` prompt rather than a "malformed list" error. When that happens we can enter `<esc>` to return to the Command: prompt. In either case the code cannot be run until the missing parenthesis is replaced.

extra right paren

If we load a file with too many close parentheses, we get an "extra right paren" error. Note that we also get this error if we are missing a left parenthesis:

```
(defun add-3 (n)
  + n 3))

Command: (load "add-3")
error: extra right paren
(load "add-3")
```

malformed string

This message tells us that somewhere in our file we began a string but we forgot to type the closing double quotes to end it.

This same error at the Command: prompt returns a `1>` prompt. This prompt usually means that a close parenthesis is needed. But in this case a close parenthesis will not end the string because it just becomes part of the string. In other words, if a close parenthesis doesn't eliminate a `1>` prompt, it means we have a malformed string. The simplest solution is to enter `<esc>` and retype the string:

```
Command: (princ "This is a string)
1> )
1> (enter <esc>)
```

12.1.2 Errors Reported by `eval`

When `eval` is given a list as an argument, its first task is to verify that the car of the list is a valid operator. If it's not, `eval` issues an error message. `eval` errors are reported long before the function is called and have nothing to do with the function itself.

These error messages are presented with a different approach in Section 4.1.2.

bad function

The first element of any list we give to `eval` must be a symbol whose value is a function. A "bad function" error indicates that we gave `eval` something else. For example,

```
Command: (+ (3 4 5))
error: bad function
(3 4 5)
(+ (3 4 5))
```

In this example, we used parentheses to group `(3 4 5)`. Since 3 is not a valid operator, `eval` gives us a "bad function" error.

Note that the first line past the error message pinpoints the exact place where the error occurred; it's telling us that 3 is a bad function name. The line below merely indicates the larger place in the code where the offending form was placed, to help us locate it in our file.

A forgotten space can also lead to this error:

```
Command: (+3 4 5)
error: bad function
(+3 4 5)
```

The operator, `+3`, is unrecognized by AutoLISP. We must separate the list elements with one or more spaces.

A "bad function" error can also arise from the misuse of `quote`. The following code may look correct, but we forgot to `quote` the list:

```
Command: (car (3 2))
error: bad function
(3 2)
(CAR (3 2))
```

As in a previous example, `eval` flags 3 as being an invalid operator.

Dotted pairs can never be operators. Forgetting to `quote` a dotted pair will also give rise to a "bad function" error:

```
Command: (setq c (3 . 2))
error: bad function
(3 . 2)
(SETQ C (3 . 2))
```

The following example shows another way a "bad function" error can arise. Suppose we accidentally redefine the symbol +:

```
Command: (setq + 6)
6
```

The next time we attempt to perform addition we get an error because the symbol + has been rebound to 6:

```
Command: (+ 3 4)
error: bad function
(+ 3 4)
```

In this case we have no choice but to exit our AutoCAD drawing and get back in, whereupon AutoLISP is reloaded and the problem corrected. When we get a "bad function" error in code that appears to be written correctly, we should not assume that it's the code that is bad; check to see whether the function has been corrupted:

```
Command: !+
6
```

null function

"null function" is a derivative case of "bad function." This message is issued when the operator is either `nil` or a symbol whose value is `nil`.

A "null function" error might arise if we use parentheses for grouping. If the value of the symbol `a` is `nil`, using it as an operator will cause a "null function" error:

```
Command: (+ (a b c))
error: null function
(A B C)
(+ (A B C))
```

bad formal argument list

"bad formal argument list" is another derivative case of "bad function." This message is issued when the operator is a symbol whose value is either a list or a cons.

As with "bad function," "bad formal argument list" can arise from misplaced parentheses. Suppose that the value of the symbol `lst` is a list or a cons. Then the following code is erroneous:

```
Command: (car (lst))
error: bad formal argument list
(lst)
(car (lst))
```

The derivation of this error message isn't of paramount importance, but "bad formal argument list" is such a strange term that understanding its origin might help us avoid the error. The *AutoLISP Reference* uses the term *arguments* to mean both *arguments* and *parameters*. Similarly, AutoLISP uses the phrase *argument list* when referring to the *parameter list*.

Functions that we write are stored as lists in the computer, like so:

```
(defun foo (a)
  (+ a 2))
```

```
Command: !foo
((A) (+ A 2))
```

The first element in the stored function list is the parameter list. When our operator is a symbol whose value is a list, AutoLISP believes we're trying to build a function and complains that it doesn't like the formal argument list (i.e., the parameters).

Advanced Note: Inasmuch as our functions are stored as lists, providing `eval` with an operator whose value is a list doesn't necessarily cause a "bad formal argument list" error to occur. After all, we can provide a *good* formal argument list, as the following example shows:

```
Command: (setq j '(nil (+ 3 4)))
(nil (+ 3 4))

Command: (j)
7
```

12.1.3 Errors Discovered by Functions

After the code has been loaded successfully and the operator has been validated, there comes a time when `eval` gives control to the function. Recall that a function's contract encompasses not only what the function does, but also the number and object types of its arguments. If we don't supply the correct arguments, then the function will issue one of the following error messages.

bad argument type

This is probably the most common error message of all. It arises when a function expects a certain type of data as an argument and we give it the wrong kind. For example, suppose we'd like to append a number to the front of a list of numbers:

```
Command: (append 1 '(2 3 4))
error: bad argument type
(APPEND 1 (QUOTE (2 3 4)))
```

The contract of the `append` function is to take *lists* as arguments. An attempt to append something other than a list will produce this error message. If we want to append a different object type to a list, we must first make a list out of it:

```
Command: (append '(1) '(2 3 4))
(1 2 3 4)
```

Occasionally we'll supply the correct arguments, but in the wrong order. For example, we can use `substr` to find a portion of a string:

```
Command: (substr "abcde" 3)
"cde"
```

But we'd better get the order of arguments correct:

```
Command: (substr 3 "abcde")
error: bad argument type
(SUBSTR 3 "abcde")
```

Having said this, it must also be noted that several subrs *do* allow us to supply arguments in any order. In particular, the `get` functions accept a point argument and prompt in either order:

```
Command: (getdist '(3 3) "Enter second point: ")
Enter second point: 3,5.5
2.5

Command: (getdist "Enter second point: " '(3 3))
Enter second point: 3,5.5
2.5
```

Inasmuch as we should be consistent in our programming, it is suggested that we don't use the `get` functions in the latter manner.

Sometimes a "bad argument type" error appears where we think it shouldn't. For example, if the values of `x` and `y` are numbers, we might think that the following function call would return a "too many arguments" error:

```
Command: (print x y)
error: bad argument type
```

Don't forget that the second argument to `print` must be a file descriptor, and `y` isn't one. The moral is, once again, that we must know the contract of each function.

A frequent cause of a "bad argument type" error stems from the incorrect use of the `quote` special operator. Remember that when we write code we supply *operands* that get evaluated to produce *arguments*.

When we specify operands, we must decide whether or not to quote them; the decision isn't discretionary. If we make the wrong choice, we may get a "bad argument type" error, as happens in the following example:

```
Command: (setq x 3.3)
3.3

Command: (fix 'x)
error: bad argument type
(FIX (QUOTE X))
```

By quoting x we're saying, "We want the argument to be the symbol x, not its value." Since `fix` requires a number, it rejects the argument.

Sometimes *not* quoting an operand will lead to a "bad argument type" error. We might forget to quote an unbound symbol and accidentally (and erroneously) supply `nil` to a function.

It is absolutely essential that we be aware of the arguments a function expects and the arguments we are supplying. To some extent the *operands* we provide a function are irrelevant; it is the *arguments* that the function is going to use. Don't be seduced by the appearance of the code!

too many arguments, *too few arguments*, and *incorrect number of arguments to a function*

These three error messages all mean the same thing: We did not supply the number of arguments that the function expected.

The "too many arguments" and "too few arguments" messages are returned by subrs. Functions that we write ourselves are less precise; they return an "incorrect number of arguments" error when the number of arguments doesn't exactly match the number of parameters.

Alone among the subrs, `quote` returns an "incorrect number of arguments" message rather than one of the other two.

As with the "bad argument type" error, these bugs usually occur when we don't have a complete understanding of a function's contract or we don't specify our data correctly; a misplaced parenthesis can exclude a required operand or add a bunch of unneeded ones. They also arise from unusual causes, as the following example shows:

```
(defun add-nums (/n1 n2)
  (setq n1 (getreal "Enter number: "))
  (setq n2 (getreal "Enter number: "))
  (+ n1 n2))

Command: (add-nums)
error: incorrect number of arguments to a function
(ADD-NUMS)
```

In this case we forgot to put a space after the forward slash in the parameter list. Therefore we've defined a function that takes two parameters, `/n1` and `n2`, instead of two local variables. Since slash is a legal character in a symbol name, `/n1` is a valid parameter name.

divide by zero

Division is a function that we're all familiar with, but it has some odd pathways that can lead to errors. In any computer language (as well as in standard mathematics), it is illegal to divide by zero. Although we wouldn't do this intentionally, it is very easy to divide by a variable that has accidentally been set to zero:

```
Command: (setq x (/ 2 5))
0

Command: (/ 9 x)
error: divide by zero
(/ 9 X)
```

Why is the result of the initial division zero? As with most arithmetic functions, operations performed on integers yield integer results. Instead of returning .4, the integer result is 0 with a remainder of 4. Actually, this isn't a problem with division; the function is just fulfilling its contract. The problem is really with our code. To be precise we should make at least one of the arguments be a real number:

```
Command: (setq y (/ 2.0 5))
0.4
```

If the operands are symbols, we can make an argument real with the `float` function:

```
Command: (setq a 2 b 5)
5

Command: (setq z (/ a (float b)))
0.4
```

Can't reenter AutoLISP

When a subr requests input, we are not allowed to supply an AutoLISP expression. An attempt to do so results in a "Can't reenter AutoLISP" message:

```
Command: (getint "Enter an integer")
Enter an integer: (* 3 4)

Can't reenter AutoLISP
Requires an integer value.
Enter an integer:
```

entsel and ssget are two more functions that can't call eval to evaluate a form. They also issue a "Can't reenter AutoLISP" message if we supply an AutoLISP expression:

```
Command: (entsel)
Select object: (list 3 3)

Can't reenter AutoLISP
*Invalid selection*
Expects a single object.

Select object:
```

It isn't catastrophic to supply an AutoLISP expression to these functions. They merely issue the warning and reprompt for the data that they need.

AutoCAD rejected function

AutoCAD rejects a function when it receives an argument that it can't handle. The most common cause of this error is running a get function inside of a command function:

```
Command: (command "text" '(4 4) 0.3 "" (getstring t "Enter string to place in drawing: "))
error: AutoCAD rejected function
```

Since AutoCAD doesn't "know" AutoLISP, it disqualifies the input. The way to get around this particular instance of the error is to run getstring first and assign the user's input to a variable, then supply that variable to the command function.

Other situations will cause an "AutoCAD rejected function" error. We'll get it if we try to setvar a read-only variable or supply an invalid table name to tblnext or tblsearch. It will also arise if we attempt to execute a function that interacts with the graphics display (such as the gets, the grs, and many of the ents) while a dialog box is active.

invalid dotted pair

This error arises for one of two reasons. It happens when we build a cons incorrectly:

```
Command: (car '(a .b))
error: invalid dotted pair
```

We forgot the space between the dot and b, so AutoLISP doesn't recognize it as a valid cons.

The next example is the more typical origin of this error message. In AutoCAD we might draw a circle in the following manner:

```
Command: circle
3P/2P/TTR/<Center point>: 3,3
Diameter/<Radius>: .5
```

Now let's try the same thing in AutoLISP:

```
Command: (command "circle" '(3 3) .5)
error: invalid dotted pair
```

In AutoLISP, when we specify a number that's less than 1, we *must* include the leading 0. The following code will give us our circle:

```
Command: (command "circle" '(3 3) 0.5)
nil
```

This is an example of what I call a *gotcha* . . . a minor error that hits us when we don't expect it and can be really tough to debug. In the next section we'll see several gotchas.

12.2 Gotchas!

AutoLISP's error messages are truly illuminating. Once we understand what they mean, we can solve most of our errors in a matter of seconds.

Harder errors to find are those that don't give error messages or those whose error messages just don't seem to make sense. The most difficult errors of all to track are ones that appear in code that has run correctly numerous times before. We'll look at examples of all of these gotcha errors.

Gotcha 1

It might happen that we accidentally bind the symbol `a` to the multiplication function, then compound the problem further by forgetting to quote a list. Watch what happens:

```
Command: (setq a *)
<Subr: #87e00d98>

Command: (setq b 6 c 7)
7

Command: (setq lst (a b c))
42
```

When the list `(a b c)` is accidentally evaluated (because we forgot to quote it), `a` is a valid operator because its value is the multiplication function. The arguments, 6 and 7, are multiplied and the result, 42, is assigned to the symbol `lst`.

It might be much later when we get a "bad argument type" error, because the value of `lst` isn't a list. By that time it's difficult to backtrack and figure out how `lst` became incorrectly bound. This gotcha presents a cogent argument for the use of local variables. If `a` is local, then a global setting won't affect its use within a function.

It is important that we use local variables wherever appropriate in a function. As we develop a programming style we'll find that we continually reuse symbol names, such as `num` for numbers, `lst` for lists, `pt` for points, and `ss` for selection sets. If these variables are global, when one function changes a variable's value it has repercussions on all other functions that use that variable. It's exceedingly difficult to locate a bug in a function whose data were corrupted by another function.

Gotcha 2

Gotcha 1 demonstrates why it is essential that we know when to use `quote`. Another `quote`-related gotcha occurs when we write a function correctly but call it incorrectly. In Chapter 6, Lab Exercise 4 we wrote a function called `symbolp` that tests whether its argument is a symbol. We might be puzzled by what happens when we run this function:

```
Command: (symbolp x)
nil
```

Since x is indeed a symbol, we go back to our function and scratch our heads, unable to discern our coding error. The problem is that by not quoting x when we *called* the function, the argument given to `symbolp` is the *value* of x, which is not a symbol. Our function runs correctly, but we supplied it with different data than we realized.

Incidentally, we'll mention one last time that in terms of symbols, lists, and conses, the use of `quote` isn't arbitrary; we must know when to use it and when to leave it out. Other objects, such as numbers and strings, evaluate to themselves, so quoting them has no effect. Stylistically, we never quote any object except a symbol, list, or cons.

Gotcha 3

This is yet another `quote`-related gotcha, this time involving parameters. `defun` is a special operator that requires its second argument to be a parameter list, which is a list of symbols. This list is implicitly quoted and is never evaluated. In fact, it is illegal to quote this list. Here's what happens when we do:

```
(defun test '(a b)
  (+ a b))

Command: (test 3 4)
error: incorrect number of arguments to a function

Command: !quote
3
```

Remember that the quote character is merely an abbreviation for the function `quote`. When we quote the parameter list, `(quote (a b))`, the symbol `quote` itself becomes the sole parameter. Then, when the function is called, the *global* symbol `quote` is bound to the first argument. Not only does the function issue an error message, but we also lose the all-important `quote` function and must exit and reenter our drawing.

It is likewise incorrect to quote the symbols in the parameter list:

```
(defun test2 ('a 'b)
  (+ a b))

Command: (test2 3 4)
error: incorrect number of arguments to a function
```

Recall the purpose of parameters: They are symbols that represent data to be supplied when the function is called. We don't care about parameters; we always want the data they are bound to. Therefore, we *never* quote parameters either in the parameter list or in the body of a function.

Gotcha 4

Here is a collection of several small but nasty gotchas, most of which involve symbols and `setq`. First, we must remember not to give our symbols the same name as a built-in function, like so:

```
Command: (setq list '(a b c))
(A B C)

Command: (list 1 2 3)
error: bad formal argument list
```

By rebinding the symbol `list` we've now lost the use of the `list` function. We must exit and reenter our drawing in order to have AutoLISP reloaded.

♦ ♦ ♦

A similar error arises when we give a symbol the same name as the function in which it is used:

```
(defun a (x)
  (setq a (* x 4)))

Command: (a 2)
8

Command: !a
8
```

We've now changed the value of `a` to 8 and lost the `a` function.

♦ ♦ ♦

Even simple assignments can pose problems if we're not careful. Suppose we name our symbols `1a`, `1b`, `1c`, and so forth. Watch what happens when we get to `1e`:

```
Command: (setq 1e 4)
error: bad argument type
(SETQ 1.0 4)
```

`e` is the indicator for scientific notation, and `1e` is equivalent to $1 * 10^0$, or 1.0. In this example, the first argument to `setq` is a number rather than a symbol.

♦ ♦ ♦

Remember to perform assignments. An error commonly made by new users is to perform a computation and forget to `setq` a variable to the result:

```
Command: (setq x 5)
5

Command: (* x 3)
15

Command: !x
5
```

This error can easily find its way into our programs. In Chapter 5, Lab Exercise 5 we wrote a function to calculate a tip. It first fixes its argument to the next lower full dollar amount, then calculates a 15 percent tip on that amount. What's wrong with the following version of that function?

```
(defun tip (price)
  (fix price)
  (* 0.15 price))
```

This code runs `fix` on the supplied amount but doesn't save the result back in the variable `price`. Thereafter it performs the multiplication on the full dollar amount rather than the converted number. Remember, if we don't `setq` the result, it won't be saved.

♦ ♦ ♦

Another place where we must use `setq` is with the `close` function:

```
Command: (setq fil (close fil))
nil
```

`close` always returns `nil`, thereby allowing us to assign `nil` to the variable `fil`. If we leave out the `setq`, `fil` remains bound to the file descriptor. Subsequent functions might believe that `fil` names an open file and attempt to do I/O to it:

```
Command: (print some-data fil)
error: file not open
```

This is the kind of gotcha that will appear when we aren't prepared for an error to occur and can be tricky to track down.

♦ ♦ ♦

Remember that `setq` can accept unlimited *pairs* of arguments. Here's another error involving misplaced parentheses. Suppose we've written the following function:

```
(defun foo (x)
  (setq a (+ x 2)
        b (sqrt a)
  (* a b)))
```

We load the function, then run it as follows:

```
Command: (foo 3)
error: bad argument type
(SETQ A (+ X 2) B (SQRT A) (* A B))
(FOO 3)
```

The error message shows us that the problem lies within the `setq`. A close parenthesis on the last line of the function definition should have been placed at the end of the third line instead, in order to complete the `setq`. The misplaced parenthesis made the form `(* a b)` be part of the `setq`. Since it's the fifth operand, and since the first of each pair of operands must be a *symbol,* `setq` issues a "bad argument type" error.

♦ ♦ ♦

A similar problem can arise with functions other than `setq`. Examine the following code:

```
(defun if2 (x)
  (if (zerop x))
      (setq x nil)
      (setq x (1- x)))
```

Now let's run it:

```
Command: (if2 3)
error: too few arguments
(IF (ZEROP X))
(IF2 3)
```

In the `if2` function we put our close parenthesis on the same line as the `if`, which is too early. In other words, we ended the `if` before it could evaluate the forms in its THEN or ELSE path. Since `if` requires at least two arguments, it returns the "too few arguments" error.

Gotcha 5

Here are a couple of gotchas involving spaces.

In Chapter 5, Lab Exercise 9 we wrote a function named `third` that returns the third element of a list. We might supply a five-element list in the following erroneous manner:

```
Command: (third '(abcde))
nil
```

Our problem is that instead of a five-element list, we supplied a list whose only element is the symbol `abcde`. If we don't put spaces between symbol names, AutoLISP doesn't know that we intend for this to be a list of five symbols.

We must also watch out for putting spaces where they don't belong. Suppose the value of `total` is 100. Then the following `cond` should return `all done`, right?

```
Command: (cond ((= total 100) 'all done))
nil
```

Wrong! Since a space delimits symbol names, `all` and `done` are considered two separate symbols. If we want them to be a single symbol, we should hyphenate the words: `all-done`. In this example the test succeeds, so the two operands, (`quote all`) and `done` are evaluated. The result of evaluating the final operand is `nil`, so that's what the `cond` returns.

Gotcha 6

This next gotcha can't be located by examining our code. It's one that can show up in several places, however, such as in the following `cond`:

```
(defun test-total (total)
  (cond ((< total 100) (princ "Not done"))
        ((= total 100) (princ "Done"))
        (t (princ "Error: Total is greater than 100"))))

Command: (test-total 123)
nil
```

The three tests check the value of the variable `total` and print one string if it's less than 100, a second string if it's equal to 100, and a third string if both those tests fail. We seem to have covered all our bases, but none of the strings are printed. What could be wrong with this code?

It turns out that the code is correct. However, earlier we accidentally entered (setq t nil), thereby rebinding the reserved symbol t. Since the value of t is now nil, the final test fails and the cond returns nil. It's unfortunate that AutoLISP allows us to alter the value of t; it's our obligation to remember never to do so.

The *AutoLISP Reference* specifies that certain subrs, such as getstring and tblnext, can take a "non-nil" argument. Strictly speaking, *anything* that's not nil will suffice. For example, we could issue a getstring as follows:

```
Command: (getstring 24 "Enter a string: ")
Enter a string: This is a long string
"This is a long string"
```

24 is non-nil but has no special meaning to someone reading our code. Contrarily, it appears as though it *does* have a particular purpose and therefore clouds our intentions. The definition of t is "non-nil", and we should use it wherever a non-nil argument is needed. However, in doing so we must remember not to rebind t to a different value or we can get into trouble.

Gotcha 7

This gotcha can getcha when we run the AutoCAD EXPLODE command from within AutoLISP. Suppose the value of e is the entity name of a block. We can explode the block at the Command: prompt in the following manner:

```
Command: explode
Select objects: !e 1 found
Select objects:
```

AutoCAD repeatedly issues the Select objects: prompt until we press <enter>. Given that understanding, we'd expect to explode the same block in AutoCAD by issuing (command "explode" e ""). This does indeed explode the block but might also issue an "Unknown command" error.

When called from within a command function, the EXPLODE command can only explode one block. As a result, we cannot supply the terminating null string. If we do, command attempts to find another command to run. In some cases it might even rerun the previous AutoCAD command.

This usually doesn't affect our code, but it's very disconcerting to receive the "Unknown command" message. The proper way to run EXPLODE in AutoLISP is without the closing double quotes: (command "explode" e).

Gotcha 8

This last little gotcha is an error that many AutoLISP programmers make when first getting familiar with the entity database. Suppose we draw a line from 2,3 to 4,5. There are a few ways to access its entity data, but this is *not* one of them:

```
Command: (setq e1 (entsel))
Select object: (click on line) (<Entity name: 88330508> (3.2122 3.8237 0.0))

Command: (entget e1)
error: bad argument type
(ENTGET E1)
```

What we forget is that `entsel` returns a *list,* not an entity name. Since `entget` requires an entity name, it returns a "bad argument type" error. When we run `entsel` we must remember that the entity name is the *car* of the result. If we're not interested in the pickpoint, we should run `entsel` within the `car` function:

```
Command: (setq e1 (car (entsel)))
Select object: (click on line) <Entity name: 88330508>
```

12.3 Error-Handling Subrs

AutoLISP has several subrs that give us some measure of control over functions' and users' errors.

`alert str`	Displays the string in an alert box on the screen and returns `nil`.

An *alert box* is a dialog box that pops up on the screen and remains in view until the user clicks on its `OK` button, thereby ensuring that the user sees our message. We typically employ it to send a warning to the user. Enter the following form to see how `alert` works:

```
Command: (alert "Web size is above maximum allowed")
nil
```

Alert boxes should be used sparingly; frequent application can become annoying to users.

`exit`	Aborts the currently running function and returns the message "quit / exit abort". The `quit` function works identically.

If our function detects an error from which it knows it cannot proceed, it can call the `quit` or `exit` functions. These functions provide a more graceful exit mechanism than having the function completely crash.

A practical use of `exit` or `quit` is described with the `c:pass` command in Section 9.6.

`quit` and `exit` are often used in conjunction with `*error*` in the manner shown next.

`*error*` str	User-modifiable subr that is called automatically by AutoLISP when an error occurs. It reports the error on the user's screen.

Whenever the user commits an error or cancels out of a function, AutoLISP calls the `*error*` subr to print both the error message and the offending code. This is one subr whose contract we actually *do* want to change from time to time.

Why would we want to redefine `*error*`? For one reason, suppose we write a function that turns off highlighting at the start, then turns it back on when it's done running. If an error occurs during the running of that function, highlighting will remain off. We can alter `*error*` to turn highlighting back on when an error is encountered, to ensure that the user's system is restored to its state prior to the running of the function.

When we define our own `*error*` function, we *must* provide exactly one parameter. The argument that is passed to `*error*` is the text of the error message itself. Our `*error*` routine should print this error message so that the user learns which error occurred.

The following `*error*` function prints an error message preceded by the string `"The error is: "`, then turns on highlighting:

```
(defun *error* (text)
  (princ (strcat "The error is: " text "\n"))
  (setvar highlight 1)                          ;Reset system variables
  (prin1))                                      ;Exit gracefully
```

When AutoLISP encounters an error, it sends the following message to the screen:

```
Command: (* 3 (+ 4 nil))
The error is: bad argument type
```

prin1 causes *error* to return nothing instead of a result, thereby making its appearance similar to the built-in subr. However, our redefining *error* has a major drawback. AutoLISP's *error* subr prints not only the error message but also the code where the error occurred, like so:

```
Command: (* 3 (+ 4 nil))
error: bad argument type
(+ 4 nil)
(* 3 (+ 4 nil))
```

When we redefine *error*, we lose the printout of our code, which is of paramount importance in debugging our programs. Therefore, we shouldn't redefine *error* until our code is error-free.

Once our program is completely debugged we might give it to our coworkers to use. If the user enters an <esc> during the function's execution, we may want to reset highlighting; we certainly *don't* want a lot of code to be spewed onto the screen. In this case, modifying *error* is appropriate.

When we copy a function or command from one of the trade journals, we frequently find that *error* is redefined by the author so that certain system variables can be reset if an error is encountered. In such cases we should examine the body of this *error* function to determine whether it's truly needed, and perhaps not use it until the function runs correctly. If we make a mistake typing in the function, our inability to see the code that caused the error will make it difficult to debug.

In this situation, as well as when we modify *error* on our own, there is the potential for another gotcha: What happens if our our own error routine itself has an error? Then there's no one around to report it. To be safe, we should save the value of the original *error* so we can get it back when we're done running our own version. In fact, to be truly elegant in our programming, when the need for a modified *error* has expired we should always restore the original subr. This can be done in one of two ways. The more common approach is to save away the subr in another variable before changing it, as follows:

```
Command: (setq *olderr* *error*)
<Subr: #87f01906>
```

When we have completed the task that uses our version of *error*, we can restore the original by entering

```
Command: (setq *error* *olderr*)
<Subr: #87f01906>
```

Another option is to bypass the use of *olderr* entirely. When we want to restore the *error* subr, we can simply enter

```
Command: (setq *error* nil)
nil
```

Through a miracle of AutoLISP, this also restores *error* back to the original subr even though its value is nil.

For larger programs we might choose to make our *error* handling a little more elaborate. At the start of the program we save the *error* subr and rebind it to our own error routine:

```
(setq *old-error* *error*
      *error* *my-error*)
```

We typically place our own *error* out of the way at the end of the program file. Here's an error routine similar to the one used by the DRAW program:

```
(defun *my-error* (msg)
  (if (not (or (eq msg "Function cancelled")                    ;User hit <esc>
               (eq msg "quit / exit abort")))                   ;QUIT or EXIT
      (princ (strcat "\nThe DRAW program can't handle this occurrence of a \""
                     msg "\" error.\n")))
  (setq *error* *old-error*)                                    ;Exit gracefully
  (princ))
```

The function first tests whether the user entered an <esc>, or if the quit or exit functions were called. If not, it prints the error message. Next, it cleans up after itself by restoring *error* to the original subr, then exits gracefully. In this fashion our environment is left the same as when we started.

12.4 Effective Debugging Techniques

Let's face it, none of us like to debug our programs. But if we know some basic debugging tools the task becomes much simpler.

The first three debugging techniques are just part of good programming:

1. Understand how AutoLISP works. One of the central purposes of the explanations in the first two parts of this book is to prepare us for our encounters with errors. Many errors involve a misuse of `eval` or `quote`. If we understand how AutoLISP works, we'll make fewer errors in the first place and readily discover the causes of the ones that do crop up.

2. Understand the error messages. Every error message tells us the precise cause of the problem. The code immediately following the error messages pinpoints the location of the error itself. When we fully understand what the error messages tell us, we'll stop fearing our mistakes. In fact, in many cases it is quicker to solve an error than it is to scan our code to find all the errors prior to running it. In other words, we can let AutoLISP do our debugging for us.

3. Comment the code!!! This is often left as an afterthought, then forgotten altogether. Commenting is a common courtesy that tells our coworkers how to use the program. Furthermore, if we have to come back to the code because of future errors (or changes in a new software release), we don't have to go through the code line by line to discern its actions.

   One of the hardest programs to debug is one that used to work and no longer does. It is emotionally draining to have to relearn code that we've already put to bed, but comments alleviate much of that pain.

When we discover the cause of a bug, we shouldn't search through the program to see whether there are any more. Instead, we should simply fix the bug and run the program again. Very often correcting one error will fix some others, so there's no sense to search for a problem until AutoLISP tells us we have one.

Misplaced parentheses are a main cause of user errors. Sometimes the code is correct except for a close parenthesis in the wrong place. When this happens, the error message reflects a problem caused by the paren but rarely leads us directly to the error. For example, we might get a "too many arguments" error because our parentheses encompass an additional operand. In this case we must study the code that is displayed with the error message in order to locate where we went wrong. Once again, we need to know the contract of each subr (or look it up) and understand what the error message is telling us.

When writing AutoLISP code, it's imperative that we use an editor that somehow balances parentheses for us. We nest our code too deeply to be able to reliably scan it for proper parenthesis balance.

Proper code indentations also aid our debugging efforts by making the code more readable. See if you can find an editor that automatically indents AutoLISP code. It will help you become familiar with the appearance of AutoLISP, and a misplaced parenthesis will make your code reformat in an obviously erroneous manner.

Commenting and code indentation style are discussed in Section 13.1.

Let's say the sneaky little bug has made it through our first lines of defense: The code loads correctly and runs without giving an error message. But it doesn't perform the proper action or return the correct result. What do we do next? Every error situation is different, of course, but we can take certain steps that will locate a majority of our bugs.

Many errors are caused by incorrect variable settings. The following function from Chapter 8, Lab Exercise 3, prints a string on the screen and prompts the user to type `"Y"` or `"N"`. If the user enters `"Y"`, the function returns `t`; if `"N"`, it returns `nil`.

```
(defun y-np (string / ans)
  (initget "Yes No")
  (setq ans (getkword (strcat string " (Yes or <No>) ")))
  (if (eq ans "yes")
      t
      nil))
```

Here's what happens when we run the function:

```
Command: (y-np "Do your really want to ERASE ALL?")
Do your really want to ERASE ALL? (Yes or <No>) Y
nil
```

This isn't right; the function should have returned `t`. Let's examine the value of the variable `ans`:

```
Command: !ans
nil
```

The value of `ans` is `nil` because `ans` is local to the function, so when the function is exited it loses the value it had inside. When we're done with a function we can no longer examine our local variables. The way to get around this restriction is to make our local variables temporarily global. We can do this by altering the first line of the `defun` in the following manner:

```
(defun y-np (string ; / ans)
```

By putting a close parenthesis and semicolon immediately after our last parameter, we end the parameter list and comment the remainder of the line. The slash and subsequent local variable declaration are not seen by AutoLISP, and the variable ans remains global. When the code is debugged, we simply remove the two underlined characters and ans is declared local once more.

Now let's run the code a second time and examine the value of ans:

```
Command: (y-np "Do your really want to ERASE ALL?")
Do your really want to ERASE ALL? (Yes or <No>) Y
nil

Command: !ans
"Yes"
```

Since ans was assigned a value, we'll assume that the first two lines in the body of the function executed correctly. If we still don't see the problem, we should next examine the if statement to determine why nil was returned. Let's run the if's test at the Command: prompt:

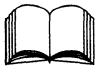
```
Command: (eq ans "yes")
nil
```

Here's our problem: The if test failed because the two strings, "Yes" and "yes", don't match. Lowercase and uppercase strings are different; they are never eq. Let's change our test string:

```
Command: (eq ans "Yes")
T
```

We'll make this change in the y-np function and test it. Once we've determined that it runs correctly we'll remove the slash and semicolon on the defun line and ans will again become a local variable.

This example employs another important debugging tool: our ability to run code at the Command: prompt. In many instances where we want to determine what a function is doing we can simply type it in. We begin with the innermost parentheses on a line and work our way outward, just as eval does. When we see the results that eval returns, we can usually locate our error.



See Section 10.3, Example 2 for a demonstration of this technique.

Sometimes making a local variable global doesn't illuminate the problem. In Chapter 6, Lab Exercise 13 we wrote the sum function to add the numbers in a list. Suppose we originally wrote it in the following way:

```
(defun sum-lst (list1 / total)
  (setq total 0)
  (foreach item list1
    (+ total item))
  (print total)
  (princ "is the sum of list elements.")
  (prin1))
```

Let's run this version:

```
Command: (sum-lst '(2 4 6))
0 is the sum of list elements
```

Obviously, 0 isn't the sum. If we make `total` global and rerun the function, we can examine `total`'s value:

```
Command: !total
0
```

We're still no closer to finding our error, so we try yet another trick: incorporating a `print` statement in our function. `print` is an exceptionally useful debugging tool because it is what we call an *invisible function:* It runs without altering AutoLISP's environment.

The contract of `print` is to take an AutoLISP object as an argument, print it on the screen, *and return that object.* Since `print` always returns the exact data it's given, it doesn't change the AutoLISP world in any way. Thus we can "wrap a `print` statement" around any code without affecting that code. `princ` and `prin1` can be used as well.

Let's go back to our function and wrap a `print` statement around the only form in the body of the `foreach`:

```
(foreach item list1
  (print (+ total item)))
```

Now we'll run the function again:

```
Command: (sum-lst '(2 4 6))
2
4
6
0 is the sum of list elements
```

Hmmm. The printout clearly shows us that the loop is accessing the list elements, but `total` isn't being set to the sum. Upon consideration we determine that this code embodies the same error that we encountered in an earlier ex-

ample: We've forgotten to `setq total` to the result, and its value is not being updated.

The `print` function is one of our most effective debugging tools, especially when an error occurs in a loop. We can "get inside" of a loop by printing a variable's value on every pass.

Another way to locate a bug in a loop (or just learn what the loop is doing) is to write the names of all the variables it uses on a piece of paper and cycle through the loop by hand. In other words, read the code line by line and write down the changes to each of the variables. This can be a somewhat tedious process, but it's the best way to become intimately acquainted with the loop's code.

Sometimes we find ourselves in an infinite (nonending) loop, or otherwise can't immediately ascertain where our code went wrong. Strategically placed `print` statements, such as

```
(print "Got here 1")
    .
    .
    .
(print "Got here 2")
```

shows us how far we get before the code bombs.

Similarly, we can use `getstring` to effect a pause so that the code doesn't just run on:

```
(getstring "Got Here 1. Hit Enter to continue.")
```

Occasionally we need to get inside of a function and stay there for a while. A *breakpoint* function causes code to stop at a specified point and print the value of a variable. We can then evaluate other forms at that point in the execution of our code. The breakpoint capability is an excellent debugging tool that, unfortunately, is absent from AutoLISP. So let's write one!

Example

`bp` takes a form as an argument, evaluates it, and prints the result. Then, in a loop, it requests other forms to evaluate. We might place `bp` within the `foreach` in the `sum-lst` function that we examined earlier, like so:

```
(foreach item list1
  (bp 'item)                          ;Breakpoint
  (+ total item))
```

Here's what appears on our screen when the function is run:

```
Command: (sum-lst '(2 4 6))
ITEM = 2

Form to evaluate: total
TOTAL = 0

Form to evaluate: (+ total item)
(+ TOTAL ITEM) = 2

Form to evaluate: (we press <enter> to resume function execution)
```

Each time through the loop the value of `item` is printed, and we are able to evaluate any other forms we choose. See if you can write this function before studying the following code.

```
1 (defun bp (form)
2   (while form                          ;User didn't hit <enter>
      (prin1 form)                       ;Print the form
      (princ "= ")
      (prin1 (eval form))                ;Evaluate form and print result
3     (prompt "\n\nForm to evaluate: ")  ;Request next form
      (setq form (read (read-line)))))   ;Get string, convert to form
```

1. The `bp` function takes a form as an argument; it's assigned to the parameter `form`. When we call `bp` (as in the `foreach`, above), we must remember to quote the operand or the form is evaluated before the function is entered.

2. The `while` loop repeats until we press the <enter> key instead of supplying a form. On each pass it prints

   - the value of the variable `form` (i.e., the supplied form),
   - an equal sign, and
   - the result of evaluating the supplied form (which is the value of the variable `form`). This is effected by the explicit call to `eval`.

3. To prepare for the next pass through its loop, `bp` prompts the user to enter another form, then runs `read-line` to input that form. `read` converts the input string into an AutoLISP expression, and `setq` binds the symbol `form` to that expression. The loop then repeats, printing the value of the latest form and requesting another. When we press <enter>, `form` becomes bound to `nil` and the loop exits.

Notes

- You might be wondering why we use `read-line` instead of `getstring` to input the form. Although `getstring` works as well, it issues a "Can't Reenter AutoLISP" error when it recognizes the form.

- In `examples.lsp` we've included a function called `bp2` that works exactly like `bp` but takes no arguments. We'd use it when we don't have a particular form that we want evaluated. A good exercise would be to try writing this function before looking at the version in `examples.lsp`.

On rare occasions even a breakpoint isn't sufficient for finding the bug. Another technique we can employ is the `trace` subr.

`trace` func . . . func	Takes one or more functions as arguments and traces entry to and exit from those functions. It returns the name of the last supplied function.

`trace` notifies us whenever a function is entered and lists the arguments that are supplied. It also prints the result when the function is exited. `trace` helps us follow the flow of a `program` and is especially beneficial when we have an error that is several levels deep in a recursive program.

In Section 6.4 we wrote a recursive function named `find-last` that returns the last element in a list. Here's a version of that function:

```
(defun find-last (elements)
  (if (null elements)
      (car elements)
      (find-last (cdr elements))))
```

Here's how the code should run:

```
Command: (find-last '(a b c))
C
```

But when we run it we get the following result:

```
Command: (find-last '(a b c))
nil
```

This won't do! Let's put a trace on that function and run it again:

```
Command: (trace find-last)
FIND-LAST

Command: (find-last '(a b c))
Entering FIND-LAST: (A B C)
  Entering FIND-LAST: (B C)
    Entering FIND-LAST: (C)
      Entering FIND-LAST: nil
       Result: nil
      Result: nil
    Result: nil
  Result: nil
nil
```

When we examine the trace of this function, we see that it does indeed sequence through all of the list elements. But it recurses one too many times and returns a final result of `nil`.

Since the recursion isn't stopping in time, the probable bug lies in the test. Indeed, on the last pass we should test whether the *cdr* of `elements` is `nil`, then return the car. By testing whether the value of `elements` itself is `nil`, the `find-last` function is called four times for a three-element list.

When the bug is resolved, we run the `untrace` subr.

`untrace` func . . . func	Turns off tracing for the specified functions and returns the name of the last supplied function.

```
Command: (untrace find-last)
FIND-LAST
```

♦ ♦ ♦

The most difficult of all bugs to squash is a *sporadic* bug: one that occurs only occasionally. An error that can't consistently be repeated is terribly hard to track down. Unfortunately, each instance of a sporadic bug is unique, so there isn't a prescribed set of steps that will always discover its source.

The possibility of a sporadic error arises when using the `osnap` function. Let's say we set it up to snap to either an endpoint or a midpoint. The user might select the intersection point of two lines, and that point might be equidistant from the endpoint of one line and the midpoint of the other. Where will it snap?

It appears to snap to the last entity we created. But if we erase the first line and redraw it, then *it* becomes the last one. So we can't determine which entity was created last with any degree of certainty. Therefore, it's best to use `osnap` in a controlled portion of the drawing where chances of conflict are minimal.

When `osnap` must be used in a crowded section of the drawing, conflict can be avoided in several ways. In Release 13 the user can have all entities that are within the aperture box highlighted cyclically by holding down the `Ctrl` key and repeatedly pressing the pick button on the pointing device. An alternative solution is to run the ZOOM command to zoom in on the selection area, then ZOOM PREVIOUS when the operation is finished.

The `osnap` function doesn't prompt for user selection, however. To avail ourselves of the preceding methods we might run a function such as `entsel` to pick up the point:

```
Command: (osnap (cadr (entsel)) "midp,endp")
(3.79534 6.83772 0.0)
```

Speaking of ZOOM, here's a seemingly innocuous piece of code. Suppose we'd like to build a selection set consisting of a line from 0,0 to 2,2. The following function will build that selection set, right?

```
Command: (setq ss1 (ssget "c" '(0 0) '(3 3)))
nil
```

Well, not always. If the entity is off the screen, `ssget` won't find it. We should have our function zoom out before building the selection set, then ZOOM PREVIOUS when done.

Sporadic bugs can't be predicted. In fact, one cropped up in an early incarnation of the `c:xdalt` command, which is Lab Exercise 11 in Chapter 10. This command requests an entity name and extended entity data, then adds the Xdata to the entity's main data. Since this bug is fairly general, we'll only examine the portion of the `c:xdalt` command that displays the erroneous code:

```
(defun c:xdalt (/ e ed)
  (setq e  (eval (read (getstring "Name of entity to modify: ")))
        id (getstring "Application ID: ")
        ed (entget e (list id)))
           .
           .
           .)
```

We run this version of `c:xdalt` as follows:

```
Command: (setq e1 (entlast))
<Entity name: 88330528>

Command: xdalt
Name of entity to modify: e1
Application ID: rh-mass
(The command requests Xdata and adds it to the entity data)
```

When the function runs, `getstring` returns the entered string `"e1"`, `read` converts it to the symbol `e1`, and `eval` returns its value, `<Entity name: 88350528>`, which is assigned to the variable `e`. The next two lines request an Application ID then get the entity data and Xdata for that application.

Every time we run this command it works exactly as expected, until we enter the following:

```
Command: (setq e (entlast))
<Entity name: 88330540>

Command: xdalt
Name of entity to modify: e
Application ID: rh-mass
error: bad argument type
(ENTGET E (LIST ID))
```

We're left scratching our heads wondering why, on the day our kid came down with the measles and our car stalled on the way to work, a program that ran correctly a thousand times before suddenly got sick too!

Given that we're quite familiar with `entget`, our first guess is that we somehow created an invalid Application ID. Let's go back to our program, make the variables global so that we can examine them when the function has completed, and rerun the command:

```
(defun c:xdalt ();/ e ed)
  (setq e  (eval (read (getstring "Name of entity to modify: ")))
        id (getstring "Application ID: ")
        ed (entget e (list id)))
           .
           .
           .)

Command: xdalt
Name of entity to modify: e
Application ID: rh-mass
(The command requests Xdata and adds it to the entity data)
```

Lo and behold, now it works! What's going on here? Since the program works when we make the variables global, we must have a problem with our local variables.

It just so happens, by wild coincidence, that outside this command the symbol we've chosen to represent our entity name is `e`, *which is the same variable name used inside the command.* As before, `getstring` returns the entered string `"e"`, `read` converts it to the symbol `e`, and `eval` returns its value. But this time the value of `e` is `nil`, not an entity name, because the *local* variable `e` is used and it is currently unbound. Thus, `e` is assigned a value of `nil` and the subsequent call to `entget` fails.

This error has a distinct possibility of arising whenever we explicitly evaluate a parameter because we don't want to limit what the user may enter. One way to protect against such a bug is to select a parameter name that the user has a slim chance of choosing. For example, if instead of `e` we named that symbol `e%%`, there's a very good chance that this bug would never occur.

This brings us to the end of our exploration into AutoLISP errors. It is hoped that seeing the causes of some common and not-so-common error conditions will better prepare you for the errors you may encounter when programming. The most important point that we made in this chapter is that the *very first form* following an error message *always* pinpoints the exact cause of the error. Keep that in mind and watch your parentheses, and you'll solve 90 percent of your errors very quickly. Understanding `eval` and good old practice will prepare you for most of the rest.

13

Programming Style

Topics Covered in This Chapter

- Making our programs more readable
- Global variables revisited
- Error-proofing code
- Writing larger programs
- Programming the AutoCAD help facility
- Understanding other programmers' code

Goals for This Chapter

- Understand the benefits and process of making our code easily understandable through proper commenting practices. Learn techniques for writing cleaner and more efficient programs.
- Delve deeper into the correct ways of using global variables.
- Learn steps we can take to make our routines robust so that they don't crash when others run them.
- Learn how to organize multiple functions into a well-written program. Study issues that we should consider when writing large programs.
- Learn how to create help files for our own AutoCAD commands.
- See how we might take someone else's code that we must maintain and rewrite it in a more organized and understandable fashion.

Introduction

As we become more experienced with AutoLISP we gain certain realizations and lightbulbs click on. Some of these make us exclaim, "Why didn't I think of that sooner?" Other insights are more profound and may take weeks or even months to realize. In this chapter, as throughout the book, my intent is to help shorten that learning curve.

In many cases AutoLISP offers several different ways to perform the same task. When we write our code, we should seek to accomplish the following goals, in order:

1. Get the program to run.

2. Make the program clear and understandable.

3. Have the program run in an efficient manner.

Clarity is pointedly placed above efficiency in our list of goals. The programs we write will all execute exceedingly fast. Besides, if speed is our primary goal we have the capability of writing the code in C and compiling it.

As we write a program, we should think about how to make it clear enough for someone in our group to pick it up and understand it . . . and for us to easily follow it when we come back to it months later. We want to avoid sinking a lot of time into relearning code that we once understood.

In this chapter I examine various steps we can take to make our code more understandable. I also discuss some programming techniques that enable the code to run more efficiently as well.

13.1 Writing Readable Programs

When we write a function using an editor, we should place the (`defun` in column 1 and indent the second line two or three spaces. Don't tab over because the `<tab>` key indents too far; large programs will have many lines that reach the right margin and wrap around, thereby making the code unreadable.

In our examples and lab solutions we've adhered to good indentation policy. Forms at the same level of parenthesis nesting line up one underneath the other. By a quick visual examination of the code we can easily discern the operands for a given function.

For a special operator, such as `if`, we usually put the operands on separate lines:

```
(if (eq ans "Yes")
    t
    nil)
```

This code clearly shows the three operands to `if`.

Many AutoLISP programmers like to place a close parenthesis under its matching open parenthesis, which is especially helpful with larger functions. For example, let's look at the `c:long-dist` command that we wrote in Section 6.3:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                          ;;;
;;;   The long-dist command repeatedly asks the user to insert a point, then ;;;
;;;   calculates and prints the distance between the new point and the       ;;;
;;;   previous one. When the user hits <enter> to complete input, it prints  ;;;
;;;   the total distance traversed and the average per leg.                  ;;;
;;;                                                                          ;;;
;;;   Original command by Bill Kramer, Kramer Consulting, Inc.               ;;;
;;;   Revised by Roy Harkow, Roy Harkow Associates, Revision 3               ;;;
;;;                                                                          ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun c:long-dist (/ tot-dist cnt p1 p2)
  (initget 1)                                          ;Disallow <enter> first pass
  (setq tot-dist 0.0                                   ; Initialize variables
        cnt      0                                     ; and get first point
        p1       (getpoint "\nStarting point (ENTER when done): ")
  ) ;end setq
  ;;Request next point, calculate and print distance, then add to tot-dist
  (while (setq p2 (getpoint p1 "\nNext point: "))      ;Next point entry
    (princ " Distance: ")                              ;Display distance during calc.
    (setq tot-dist (+ tot-dist (prin1 (distance p1 p2)))  ;Add distance traveled to total
          cnt      (1+ cnt)                            ;Increment leg counter.
          p1       p2                                  ;Bump P1 to P2's location
    ) ;end setq
  ) ;end while
  ;;If cnt is greater than 0 print the total distance traversed,
  ;; then calculate and print the average.
  (and (> cnt 0)                                       ;At least one point entered?
       (princ "\nTotal traversed distance: ")          ;Yes
       (prin1 tot-dist)                                ;Total distance traveled.
       (princ "    Average: ")
       (prin1 (/ tot-dist cnt))                        ;Average
  ) ;end and
  (prin1)
) ;end defun
```

In this listing we can clearly see where the `setqs`, `while`, and `and` end. We've not only aligned each close parenthesis under its matching open, we've also described the structures they're closing with comments. In most cases this is overkill, but it can help clarify more complex code. Furthermore, we've made the `setqs` more readable by aligning both the first *and* second operand in each pair.

I've mentioned on several occasions that commenting our code is of utmost importance. Comments aren't loaded with the function, so they don't take up space in memory. Thus, there's no reason to *not* provide comments. It also behooves us to adopt a rigorous commenting style so that our functions all have a uniform appearance and are easier to follow.

Before each function we should provide a detailed explanation of the function's contract: how many and what kind of arguments the function takes, what it does, what it returns. List the global variables and explain the purpose of each. Indicate how to run the function and any peculiarities in doing so. We might also list the functions called by this one, and what those functions do. Finally, if we're sharing this function with other people, it's probably worthwhile including our name, the date, and the current revision number.

Within the function we should try to comment as many lines as possible, especially those whose meanings aren't immediately apparent. Here are some commenting conventions that many AutoLISP programmers have adopted:

- Use three semicolons prior to a comment block at the beginning of a function. Start the semicolons in column 1.

- Place two semicolons before a comment that explains a block of code, such as the ones before the `while` and the `and`. Align the semicolons with the code itself.

- Put one semicolon in front of a comment that is on the line of code it's describing. Where possible, in-line comments should line up underneath each other in a column. If a comment is a continuation of the line above, indent a couple of spaces after the semicolon.

- When putting a close parenthesis on its own line, optionally add a comment that specifies the name of the function that it's ending.

If we wish to comment a large block of code we can precede the comment with a semicolon and vertical bar pair (`;|`) and conclude the comment with a verti-

cal bar and semicolon pair (| ;). This obviates the need for placing semicolons on every comment line.

This technique comes in especially handy when we want to temporarily comment out a part of a function, or comment an entire function that's in a file with many other functions. Rather than having to place semicolons on each line, we can simply add ; | and | ; pairs and the code will not be loaded by AutoLISP.

13.1.1 Using Global Variables

When I discussed global and local variables in Section 5.2, I emphasized that in a vast majority of situations local variables are preferable to globals. In a few circumstances, however, we need to use global variables.

Since local variables go away when a function is finished, we can't inspect their values at the Command: prompt. Therefore, when debugging code we might want to temporarily make our local variables global. As we've previously seen, we can accomplish this by putting a close parenthesis and semicolon immediately after our last parameter:

```
(defun y-np (string); / ans)
```

When the code is debugged we simply remove these two characters and `ans` is declared local once more.

In a more permanent sense, we use globals when our program needs to remember the value of a variable after the function concludes. For example, in Chapter 10, Lab Exercise 12 we wrote functions called `c:unblk` to explode a block and `c:reblk` to reform the block. The `c:unblk` command uses global variables to remember the name and starting point of the block so that it can be rebuilt by the `c:reblk` command.

Another place where globals come in handy is with prompts. When our AutoCAD commands prompt the user for input, we frequently want to supply the user's previous response as a default, much the same way as the TEXT command prompts with the previously entered height when it asks for a new text height.

Suppose our program requests the name of a room to work on and saves that value in the global variable `*room*`. The next time the user calls the function we supply the user's last response in the prompt. Here's one way to accomplish this:

```
(if *room*
  (setq *room* (getstring (strcat "\nRoom to work on <" *room* ">: ")))
  (setq *room* (getstring "\nRoom to work on: ")))
```

The code must first test whether `*room*` is bound because `nil` is an invalid argument to `strcat`. If it has a value, we concatenate the following three string segments into one long string:

- `"\nRoom to work on <"`
- The value of `*room*`
- `">:"`

We must use `strcat` rather than a supplying a single string because we want the variable `*room*` to be evaluated. Note that if the user presses <enter> to accept the default, `*room*` will be bound to the null string. Our code must take this possibility into account.

Using global variables can be dangerous. If we happen to give a global variable the same name as a global variable in another function, one variable's value will probably overwrite the other's. To minimize the opportunities for conflict we give our global variables unusual names.

A Common Lisp convention is to put asterisks on either side of a global variable's name, as Autodesk did with `*error*` and we did in the last example with `*room*`. By always doing so, however, we increase the chances for conflicts in our own code because the convention removes the uniqueness in our variable names. Because a chance of conflict always exists, it's essential that our prefunction comment block list the names and purposes of all global variables used in that function.

♦ ♦ ♦

Stylistically, we never end a function with a `setq`. We don't often use global variables and, when we do, there's rarely a need to bind one at the end of a function. Recall the `dtr` function that converts degrees to radians. We *could* write it in the following manner:

```
(defun dtr (deg)
  (setq *r* (/ (* pi deg) 180.0)))
```

If someone (probably ourselves) is using the variable `*r*` to remember, say, the radius of a circle, an invocation of `dtr` overwrites the value of that variable.

Admittedly, the chances of this happening are slim, but why take a risk at all? The proper way to set this variable is outside the function, as follows:

```
(defun dtr (deg)
  (/ (* pi deg) 180.0))

Command: (setq r (dtr 180))
3.1415
```

Since `dtr` returns the result, it's safer to set it to a symbol when we call the function than inside the function itself. Furthermore, this method is more flexible: The user can set the result to the variable of her choosing instead of always having it set to `*r*` as in the first example.

Another subr we almost never put on the last line of a function is `print` (and its derivatives). We write functions that return results instead of printing them; the read-eval-print loop prints those results for us. If the last form evaluated in a function is a `print` statement, the result is printed twice. The one exception is that we often end our AutoCAD *commands* with a `princ` or `prin1` with no arguments, which causes the command to exit gracefully and return nothing.

Can we `setq` *local* variables in the last line of a `defun`? Since local variables disappear as soon as the function finishes, we *never* have a reason to do this. It's not wrong to `setq` a local variable on the last line of a `defun`, merely superfluous.

Incidentally, in its description of `defun` the *AutoLISP Reference* states that it is possible to define more than one local variable with a given name, and it presents the following example:

```
(defun fubar (a b / a a b)
```

In this case the first instances of `a` and `b` are the only ones used; the others are ignored. Since there is no call for doing this and it's quite confusing as well, we should *never* define the same variable name twice in a given function.

♦ ♦ ♦

AutoCAD provides us with the capability of implementing "super global variables," variables whose values are maintained across drawing sessions. There are 15 system variables that can store data and retain that information after the drawing has been closed. These variables are as follows:

USERI1 to 5	Store integer values
USERR1 to 5	Store real values
USERS1 to 5	Store string values. Maximum string length is platform-dependent

We would use these variables as follows:

```
Command: (setvar "userr3" min-dist)
0.5
```

In this example `min-dist` is bound to a minimum distance that was entered by the user. At the end of the drawing session we want to remember this distance so we don't have to reprompt the user next time. We could write this information to a file, but it's much simpler to store it in a system variable. The next time we enter our drawing we can `getvar` USERR3 to recall the distance.

`setcfg` and `getcfg` provide an even more powerful way of saving data across drawing sessions. They are Release 13 functions.

`setcfg` str str	Takes strings specifying the name of a variable in the `acad.cfg` file and a value, and assigns the value to the variable. It returns the value.

The `setcfg` subr allows us to write data to the AppData section of the `acad.cfg` file. The information that we write to this file outlives the drawing session and can be read when we are in a different drawing. This means that we no longer have to create files just to pass information between drawings.

`setcfg` writes to the `acad.cfg` file in either the `acadr13\dos` or `acadr13\win` directory. The first section, labeled `"AppData"`, is where we can write data for our application. We normally write data into an area set up for our specific application, and that area can be further subdivided into smaller sections.

If we're drawing office buildings, for example, we might want to save information for the architectural, piping, and electrical applications. Furthermore, we'd like to subdivide the architectural information by floor. We could save the number of windows on the second floor as follows:

```
Command: (setcfg "appdata/arch/floor2/windows" "28")
"28"
```

The first argument to `setcfg` is a string that specifies the section where we want our data written. We can think of the AppData area as a directory structure and write to whatever "subdirectory" we choose. If we write to a section that doesn't exist, `setcfg` creates that section for us. The minimum string we can supply to `setcfg` is `"appdata/"` followed by one character; the maximum length string is 132 characters.

The second argument to `setcfg` is a string that represents the data we want written. Its maximum length is 347 characters.

If we specify the AppData string incorrectly, `setcfg` either issues a "bad configuration variable name" error or returns `nil`. If `setcfg` succeeds, it returns the second argument.

`getcfg str`	Takes the name of a variable in the `acad.cfg` file and returns its value.

`getcfg` is the companion to `setcfg` and follows all of the rules just described. To obtain the value that we set earlier we would run `getcfg` as follows:

```
Command: (getcfg "appdata/arch/floor2/windows")
"28"
```

♦ ♦ ♦

At the beginning of an AutoCAD command, AutoLISP programmers frequently alter system variables whose values might have been set by the user. For example, many of us like to set HIGHLIGHT to 0 in order to turn off highlighting. If we're courteous programmers, we'll turn highlighting back on at the end of the function.

Herein lies a problem: Suppose the user really wants highlighting turned *off*. By restoring this feature we're corrupting the environment anyway. If we want to be truly elegant programmers, we should save away the value of any system variable that we change at the beginning of the function and restore that value at the end:

```
(defun c:test (/ hlt)
  (setq hlt (getvar "highlight"))
  (setvar "highlight" 0)
        .
        .
        .
  (setvar "highlight" hlt)
)
```

In Section 9.4 we saw how to automate this task with the `setv` and `rsetv` functions:

```
(defun c:test ()
  (setv "highlight" 0)
      .
      .
      .
  (rsetv "highlight")
)
```

These functions only save us a line of code, so why bother? What they provide is a rigorous framework for altering and restoring system variables. Although they don't save much typing, they add cleanliness and clarity to our code. Thus, `setv` and `rsetv` are nice functions to place in our `acad.lsp` file and call within the AutoCAD commands that we write.

Regardless of which method we use for working with system variables, when our command is completed the global environment shouldn't be gratuitously altered. An elegant program leaves no footprints!

13.1.2 Error-Proofing Code

Many AutoLISP programmers discover that much of our programming involves writing time-saving AutoCAD commands for members of our company to use during drawing sessions. Once users have these commands they're beyond our control. Keep in mind Murphy's original law: "What can go wrong will go wrong." As good AutoLISP programmers we should eliminate beforehand as many error possibilities as we can.

Error-proofing our code involves anticipating how our coworkers might use our commands, what invalid keystrokes and mouse clicks they could possibly enter, plus any other ways they might sabotage our noble attempts at increasing their productivity. Once we've thought of these possibilities we should incorporate the remedies into the code. If the program is large enough and important enough, *beta test* it: Give it to some coworkers to try for a few days and see what bugs and barriers they encounter.

One problem that we must guard against is the user entering an `<esc>` during function execution. Normally when that happens AutoLISP prints a "Function cancelled" message on the screen followed by numerous lines of aborted code. This can be quite unsettling to someone not initiated into the ways of AutoLISP and instill a fear of using our commands in the future.

For this reason it pays to test for an <esc>, then print a gentler message or exit quietly. In Section 12.3 we showed how to modify the *error* function to handle this occurrence. If we alter *error*, the offending code is not dumped to the screen following an error message. The corollary is that we should only change *error* when our command is totally debugged and we're ready to share it with our coworkers.

♦ ♦ ♦

We typically *don't* check a function's parameters to verify that the correct data was entered. Recall the add-em function:

```
(defun add-em (n1 n2)
  (+ n1 n2))
```

If the function is called with a value that is not a number, then the addition function reports the bug:

```
Command: (add-em 3 nil)
error: bad argument type
```

Since the error is uncovered by the addition function, we don't need to test for a bad argument every time the function is run. That would just add extra work for the vast majority of function calls for which the input is correct.

On occasions when we can't allow a command to crash we may have to control the input. In most cases the get functions should be used to obtain data from the user. Although the gets provide decent control over the kind of data that our command will accept, even they can allow erroneous input to get through. For example, in Section 8.1.1 we wrote a function that sets initget bit 128 to allow for arbitrary string input:

```
(defun getpnt (/ pp)
  (initget 128)
  (setq pp (getpoint "Select a point: "))
  (if (eq (type pp) 'str)
      (eval (read pp))
      pp))
```

This function permits the user to enter a symbol whose value is a point instead of entering the point itself. But if the user enters (3 4), it's not taken as a point. Instead, eval attempts to evaluate the list and, since 3 isn't a valid operator, returns a "bad function" error. To be truly robust, this function might scan a list of acceptable strings before attempting to evaluate the input.

Our functions receive data from the users in other ways, such as through `entsel`. If the user misclicks and doesn't select an entity in response to its Select object: prompt, `entsel` returns `nil`. An assumption that we have a list after running `entsel` is a seminal step in the birth of a bug.

If we can't afford to have `entsel` return an error message, we must provide code to get around a possible misclick. The simplest remedy is to write a loop that re-prompts if `entsel` returns `nil` or the user selects the incorrect entity type:

```
(setq e (car (entsel "\nSelect circle: ")))
(while (or (null e)                                                    ;Nothing selected or
           (not (eq (cdr (assoc 0 (entget e))) "CIRCLE")))             ;  not a circle
  (setq e (car (entsel "\nEntity selected is not a circle. Select again: "))))
```

Another fault line runs through the realm of file I/O. When we ask the user to enter a filename, we should verify that the file exists before trying to open it. One way to test for a file is with `findfile`, but, remember, `findfile` searches the AutoCAD library path for the file. It might return a file with the same name that's in the wrong directory. In most cases it's safer to use `getfiled` to obtain a filename from the user.

There are many different possibilities for user errors, but not that many different *kinds* of possibilities. In other words, once we gain familiarity with the causes of errors we can generally plan for and avoid nearly all of them. The key is to think through the program ahead of time and consider the possibilities.

♦ ♦ ♦

Another way we can make our code robust is to prepare for our own errors. One way to do so is to keep our code as general as possible, such as by using variables instead of constants.

In Section 10.3 we studied a function named `boxit` that draws a box around some text. `boxit2` is a version that uses the PLINE command instead of `entmake`. It also assigns an offset of .2 to `off` rather than requesting an offset from the user.

```
(defun boxit2 (p1 p2 / off)
  (setq off 0.2)
  (command "pline"
    (list (- (car p1) off) (- (cadr p1) off))       ;Lower left
    (list (- (car p1) off) (+ (cadr p2) off))       ;Upper left
    (list (+ (car p2) off) (+ (cadr p2) off))       ;Upper right
    (list (+ (car p2) off) (- (cadr p1) off))       ;Lower right
    "c"))                                           ;Close pline
```

The offset amount could easily have been hard coded into the function (thereby eliminating the need for the variable `off`), but that would create problems if the offset ever changed. The present method enables us to alter the offset amount with a single change instead of searching through the code and making multiple revisions. The more changes we have to make, the more tedious the process becomes, and the more chances there are that we'll make an error.

Symbolic representation of our constants is even more important when we write large programs. Suppose we draw a border whose corner points are 0,0 and 8,8 around our drawing. Perhaps we'll place a table one unit below and one unit to the left of the upper right corner, at 7,7. If the border changes, that table might no longer be in the right position. Multiply that by dozens of entities created in the same manner and we can envision a major headache.

We can protect ourselves by storing the border's corner point, 8,8, in the variable `pt2`. Then we would specify the coordinates of the table as

```
(list (1- (car pt2)) (1- (cadr pt2)))
```

This might be a little cumbersome but, if we've created a large database in AutoLISP, when the time comes to change our constants we'll be very happy we put in the effort to make our code general.

♦ ♦ ♦

When a new version of AutoCAD comes out we sometimes find that our AutoLISP routines must be revised to fit the changed environment. If some of our coworkers are using the new release and others are using the old one, or if we're using alternate versions on different days, both copies of these programs must be available. Furthermore, we need to have a way of accessing the version that we need. The `ver` function provides the means for doing that.

`ver`	Returns a string containing the current AutoLISP version number.

We can run `ver` at the Command: prompt as follows:

```
Command: (ver)
"AutoLISP Release 13.0 (en)"
```

How does this pertain to our files? Suppose `border12` is a function that draws a border in Release 12, and `border13` is its Release 13 counterpart. We can

write a simple "prefunction" to determine which AutoCAD version is currently running, then call the appropriate function, as follows:

```
(defun c:border ()
  (if (equal (ver) "AutoLISP Release 13.0 (en)")
      (border13)
      (border12))
)
```

Now the proper function is called automatically and the user doesn't even have to consider what version of AutoCAD is running.

13.1.3 Tips and Techniques

We pick up a variety of little tricks after programming for a while in AutoLISP. But why wait when we can start using these now?

Tip 1

If you're new to AutoLISP, make ample use of `setq`. For example, the following function adds an AutoLISP object to the end of a list:

```
(defun add-end (l1 elt / l2 l3)
  (setq l2 (reverse l1))                ;Reverse the list
  (setq l3 (cons elt l2))               ;Add the element to the front
  (reverse l3))                         ;Reverse it back
```

Once we become more familiar with AutoLISP, we typically combine these three steps into one line of code, as follows:

```
(defun add-end (l1 elt)
  (reverse (cons elt (reverse l1))))
```

It's cleaner to write code the latter way, but when we're just starting it's easier to understand the former. Don't try to do too much in one step; we don't get bonus points for brevity!

When we do nest functions in this manner, it's important to make them readable. Proper use of spaces helps. As far as AutoLISP is concerned, the following function is identical to the last one, but to us it's not nearly as nice:

```
(defun add-end(l1 elt)
(reverse(cons elt(reverse l1))))
```

Within nested functions we can place a `setq` wherever we want. Its second argument is assigned to its first and returned, but nothing else is changed in the AutoLISP world. Thus, we can "wrap a `setq`" around any expression without changing it. For example, let's say that in the `add-end` function we'd like to save the reversed list. Then we could revise the function as follows:

```
(defun add-end (l1 elt)
  (reverse (setq l2 (cons elt (reverse l1)))))
```

The reversed list is assigned to `l2` and returned by `setq`, whereby it is reversed back to its original order and returned by `add-end`.

We can put `setq` to use when debugging our code as well. Suppose we've written the following function:

```
(defun very-long-function-name (lst)
            .
            .
            .)
```

When debugging this function, we don't want to type its entire name every time we run it. To save keystrokes, we'll bind the symbol `l` to a list and the symbol `v` to the function itself:

```
Command: (setq l '(1 2 3 4 5))
(1 2 3 4 5)

Command: (setq v very-long-function-name)
((/ LST) ...)
```

Now we can quickly invoke the `very-long-function-name` function as follows:

```
Command: (v l)
```

We can simplify this process even further by making `v` a command:

```
Command: (defun c:v ()
1> (very-long-function-name l))
C:V
```

Now we can test `very-long-function-name` on the list we created by simply typing `v` at the Command: prompt:

```
Command: v
```

If we have to run this function multiple times in order to remove the bugs, it saves a measurable amount of time to not have to repeatedly enter the entire function name and all of the data.

Tip 2

The next tip is one of AutoLISP's main benefits: We can create a series of short functions to optimize actions we perform repeatedly. For example, suppose we constantly go through our drawing and change the text on the screen. We can automate this with a simplified change text command:

```
(defun c:ct (/ ed s)
  (setq ed (entget (car entsel)))                    ;Get edata for chosen entity
  (setq s (getstring t "New text: "))                ;Get new text
  (setq ed (subst (cons 1 s) (assoc 1 ed) ed))       ;  and insert it into the data
  (entmod ed)
  (princ))
```

Here's how this command looks to the user:

```
Command: ct
Select object: (User clicks on text)
New text: abc  (Original text is replaced by "abc")
```

Our aim is to eliminate keystrokes and mouse clicks; the fewer we need to do, the more rapidly we can draw. Certainly we can accomplish the same task with DDEDIT, but `c:ct` is faster.

Here's another example. Perhaps our current project calls for us to repeatedly window one or more entities and copy them from an endpoint of one to the midpoint of some other entity. We can speed up this task with the following one-liner:

```
(defun c:cop (/ c)
  (command "copy" "w" pause pause "" "end" pause "mid" pause))
```

This command pauses for the window coordinates and performs the proper snaps. Again, this routine in and of itself doesn't save a tremendous amount of time, but after using it 50 or 60 times we'll be very glad we have it. We can even write code to perform a partial command:

```
(defun c:mc (/ c)
  (command "move" "c" pause pause "" "int" pause))
```

Here we select entities in a crossing window and move them from the intersection of two lines to . . . well, we let the user decide that. But even a short savings such as this adds up in the long run.

Don't forget that we can simplify our repetitive AutoLISP tasks as well. The following command automates our retrieval of the data for an entity selected by the user:

```
(defun c:sel ()
  (entget (car (entsel))))
```

Tip 3

Quite a few functions return `nil` when they are done. These are excellent candidates for tests to a `while` or an `if`. For example, since `nil` is an end-of-file indicator, we can read lines from a file in the following manner:

```
(while (setq txt (read-line fil))
  . . .)
```

Each time through the loop the next line is read from `fil` and assigned to the variable `txt`. When the end of file is reached, `read-line` returns `nil`. The result is propagated through the `setq` to the `while`, and since the test returns `nil`, the loop is exited. In this way we've made the test do meaningful work.

The `c:printit` command in Section 8.3 demonstrates this technique.

The `get` functions can be used in a similar manner if we tell the user to press `<enter>` when done inputting data. An exception to this is `getstring`, which returns the null string when the user presses `<enter>`. When using `getstring` as the test, we must specifically check for the null string.

Another place we use this technique is with `entnext`. We can sequence through our entity names in order; `nil` is returned when the end of the entity database is encountered.

Tip 4

Sometimes we want to perform an action if the value of a variable *isn't* `nil`. We can test for `nil` with the `null` predicate:

```
(if (not (null x))
    . . .)
```

If the value of x is nil, the null function returns t, not returns nil, and the ELSE path is taken. If, on the other hand, the value of x is not nil, the null function returns nil, not returns t, and the THEN path is taken. What we have here is a double negative that cancels itself.

We can ask this same question using the boundp function:

```
(if (boundp 'x)
    . . .)
```

If x is *bound,* meaning its value is non-nil, the test succeeds and the THEN path is taken.

We can write this code in an even more straightforward manner without using either null or boundp:

```
(if x
    . . .)
```

As with the previous two examples, this if test succeeds if x is bound. Whereas the null predicate is useful in other applications, this is the only situation in which boundp would be used. The latter approach renders the boundp function superfluous.

13.1.4 Writing Larger Programs

One question that often arises is, should a file contain one function or many? The answer depends on the application.

If we write a command that stands on its own, we might wish to keep it in its own file. Space is saved in memory if we only load functions and commands when we need to use them. If a file contains a single command, it's beneficial to give that file the same name as the command. For example, we should place the command c:sum in the file sum.lsp. If it's in the file add.lsp, we have two names to remember instead of one, and we could become confused.

A more elaborate program will comprise several functions. Since these functions are all used together, they should be placed in the same file so that they can all be loaded at once. Give the file a name that reflects what the program is doing. If a main function or command is always called first, give the file the same name as that routine. For example, the c:draw program that draws cons diagrams is composed of numerous functions and stored in the file draw.lsp.

Incidentally, it's worth repeating here that functions, commands, and variables should also be given names that reflect their purpose. Without even reading its code we have an inkling of what the `c:sum` command does, which is beneficial when we need to scan our directories to find it. Furthermore, the variable where this command stores its results is called `total`. `q` or `pt2` would work as well, but the former symbol is meaningless to us and the latter (which typically represents a point) is confusing.

As we become more experienced we tend to write larger, more extensive programs. However, AutoLISP programmers still strive to keep their functions brief. Shorter functions are easier to debug, simpler to maintain, and more easily understood when we come back to them at a later date. Furthermore, our programming is facilitated when we can see an entire function (or more) printed on our screen.

When we write a program, there is no need to enter all of the code at one time. Type in a few lines, debug them, then add a few more. If we're unsure whether a line of code is correct, we can type it at the Command: prompt and watch it run.

Wherever possible we extract several lines from a large routine and make them their own function. The extra time it takes AutoLISP to process an additional function call is completely negligible, and it's much easier to follow the flow of a few small functions than a single long one.

How do we know when to split a long routine into several shorter ones? Any code that is repeated in a routine is a good candidate for becoming its own function. A code segment that stands on its own and performs a modular task may also be extracted. For example, consider the `zip` function that we wrote in Chapter 9, Lab Exercise 5 to test the validity of a zip code string:

```
(defun zip (code)
  (cond ((= (strlen code) 5)                ;If 5 chars
         (check code))                      ;  test it
        ((= (strlen code) 10)               ;If 10 chars
         (and
           (check (substr code 1 5))        ;  test first 5
           (check (substr code 7 4))        ;  then next 4
           (= (substr code 6 1) "-")))      ;  and hyphen between
        (t nil)))                           ;Otherwise, not valid

(defun check (code)
  (eq (type (read code)) 'int))             ;Is this a number?
```

On three occasions `zip` must test whether a string represents an integer. It calls the `check` function three times rather than executing the same code repeatedly.

A related decision that we must make for a larger program is which routines should be commands and which should be functions. Commands are nice for communicating with the user because they present an interface with which the user is familiar. Functions are easier for programmers to work with because they allow us to pass data via the parameters.

The typical solution is to make the first routine be an AutoCAD command. This routine (which is analogous to the *main* routine in other programming languages) initializes some key variables and asks the user to enter the data that the other functions will use.

In many cases this command is full of `get` functions and `setq`s. When our code performs multiple assignments, we typically place them all in one `setq`. Here are the first few lines of the `c:xdalt` command that we wrote in Chapter 10, Lab Exercise 11:

```
(defun c:xdalt (/ e id ed l1 l2 l3 typ data)
  (setq e (eval (read (getstring "Name of entity to modify: ")))
        id (getstring "Application ID: ")
        ed (entget e (list id))
  )
```

Once the variables are initialized, headings printed, and basic structure drawn, the command can call one or more functions to perform the actual work. These functions are called *subroutines* in other programming languages. In AutoLISP any function can call any other, so the concept of a subroutine has no real meaning to us.

♦ ♦ ♦

An important point that was made much earlier in this book is worth repeating and emphasizing here: When a file is loaded into memory, everything in that file is evaluated as though we had typed it at the Command: prompt.

One implication of this is that a file can contain AutoLISP functions *outside* of `defuns`. These functions are run as a script when the file is loaded. In this light it's quite nice to place (`princ`), outside of a `defun`, on the last line of a file. This causes `load` to return *nothing* instead of the name of the last function defined. For example, nothing is returned when the `examples.lsp` file is loaded because it ends with a (`princ`).

Whereas it's also elegant to end the `acad.lsp` file with a (`princ`), we usually don't leave other loose forms lying about. Single functions that we want run as a script, such as `getvars` and `commands`, are placed inside of the `s::startup` function so that they are certain to be executed. It also makes the `acad.lsp` file cleaner.

When a file is loaded, all of its `defuns` are evaluated. This means that the functions are defined, given a name, and brought into the AutoLISP world so they can be run at a later time; they are *not* executed when the file is loaded. For this reason the order of functions in a file is irrelevant to AutoLISP. For people reading our code, however, the order in which we place our functions is very important.

Functions can be ordered in a top-down manner. The functions called by the top-level command are placed first, functions called by those functions come next, and so forth. At the very end of the file we put the functions that perform the lowest-level tasks, such as drawing minor components of larger structures or setting up a database.

This isn't absolute. Sometimes it makes more sense to group higher-level functions with the low-level functions that do their work. The point is to have some sort of organization and not just place functions randomly in a file.

Suppose we decide to create a program to play blackjack. We could write it as follows:

```
(defun c:blackjack ()
  (create-deck)
  (shuffle)
  (deal)
  (play))
```

Now the program is finished except for some minor details! Seriously, a program can be written this way in a very logical, top-down fashion. As we work on the `create-deck` function we can define the other functions to be *stubs* so that if they're called they won't cause errors. For example, the `shuffle` function might initially be defined to be a stub as follows:

```
(defun shuffle ()
  (princ "\nSHUFFLE is not yet defined"))
```

As we get farther along in the program's creation we replace the stubs with actual code. In this way it's possible to build a fairly involved program directly in

an editor without writing it on paper first. Furthermore, if we test our code every few lines, we'll find that we've made very few errors as well.

Some programmers prefer to begin a program by writing the lowest-level routines first. The appeal of doing this is so that we know the organization of the data when we start writing the higher-level code. It is quite possible to write a fairly involved program only to discover late in the process that our data are organized incorrectly.

If we discover a fatal flaw in the definition of our data, we must first correct the data, then revise all of our code that manipulates it. In doing so we invariably miss a change or alter something we shouldn't. Errors in routines that formerly worked are among the most tiring to track down. Therefore, it behooves us to have a clear understanding of our program's contract (specifically, in this case, what the data look like) before we attempt to write the program.

Let's examine a listing that shows the relationships among the functions in the `draw.lsp` file:

```
c:draw
  choose-obj
    list?
    draw-list
      l-box
      ground
    draw-cons
      c-box
    draw-atom
      otos
      ellipse
      f-box
        box
c:run
*my-error*
```

The `c:draw` command is the main routine. It initializes variables and layers, replaces `*error*` with its own error-handling routine, asks the user for the starting point and the data to draw, and calls the `choose-obj` function to perform the actual work.

`choose-obj` (with the help of `list?`) simply determines whether the object is a list, a cons, or an atom, then calls `draw-list`, `draw-cons`, or `draw-atom`, respectively.

draw-list, draw-cons, and draw-atom perform similar tasks. draw-list, for example, causes a list (including its elements) to be drawn on the screen. It, in turn, calls l-box to do the dirty work of drawing the lines and arrows. At the lowest level the box function is called by l-box, c-box, and f-box to draw a box. Note that since the car of a cons is itself a list, cons, or atom, l-box must also call draw-list (recursively), draw-cons, or draw-atom. Thus, it's essential that we make each task modular.

c:run is a separate command that runs the code that c:draw draws. Because it's independent of all other functions (it runs after c:draw is finished), it is situated after the others. At the very end we put our error-handling routine; it's present in many of our programs and not really germane to the logic of any.

♦ ♦ ♦

Some AutoLISP coders place defuns inside of defuns, like so:

```
(defun f1 (x)
   .
   .
   .
  (defun f2 (y)
      .
      .
      .
  )
)
```

In this way f2 isn't defined until f1 is run. This may save some space temporarily, but it makes our code exceedingly difficult to understand and debug. defun is a top-level function that should *never* be nested within another function. When we need a function to be temporarily defined within a defun, we use lambda.

If we want to save space in memory (not as strong a requirement as it used to be), we can modify the acad.lsp file in several ways. For starters, we should keep acad.lsp as small as possible. Only include commands and functions that are frequently used. A large acad.lsp file not only eats up space but also takes a long time to load.

Suppose we're working on a big project that requires us to load several files during every drawing session. We could put all of the functions individually in acad.lsp, but that would make the file huge (and hard to clean up when the project is done). Instead, we can put the following function in the acad.lsp file to be run as a script:

```
(load "filename")
```

load is recursive; if it encounters another load function in a file that it's loading, it loads that file as well. When the project is finished, we can simply delete this line of code and the acad.lsp file is cleaned up. The load can also be placed inside of s::startup but, since it's temporary, doesn't have to be.

Sometimes it's hard to decide whether we should place a specific command in acad.lsp. For example, in Section 11.1 we saw a command called c:att-ed that enables us to easily modify the attributes of blocks. If we want to use this routine fairly heavily but only with certain drawings, we can save space in one of the following ways. In Release 12 we can define it in the acad.lsp file as follows:

```
  (defun c:att-ed ()
1   (load "c:/mydir/att-ed")
2   (c:att-ed))
```

1. The first time c:att-ed is called it loads the file att-ed.lsp, which resides in the mydir directory. During the loading process the defun is evaluated and the symbol c:att-ed is rebound to the newly loaded command. This is the same process as when we correct an error to a function and reload the changed definition; the function name is rebound to the new definition, thereby redefining the function.

2. When the form (c:att-ed) is evaluated, the "real" c:att-ed command is run. Thereafter it remains in memory until the drawing is exited.

In Release 13 this process is simplified by the autoload function, which performs the load for us.

autoload str list	Takes a filename and a list of strings that name user-defined AutoCAD commands and returns nil. The file is automatically loaded the first time one of the commands is invoked.

The autoload function provides the capability of having our files loaded automatically when we first need to use them. We run it as follows:

```
Command: (autoload "att-ed" '("att-set" "att-ed"))
nil
```

The first argument to `autoload` is the name of a `.lsp` file that we want loaded. The second argument is a list of AutoCAD commands that are defined in the file. The first time we call one of the specified commands, the file is automatically loaded.

In the preceding example, the file `att-ed.lsp` is loaded the first time we invoke either the `c:att-set` or `c:att-ed` command. The `autoload` works only once. If we change the loaded command, we must rerun `autoload` to have the file autoloaded again.

A command in an autoloaded file might call one of our AutoCAD commands that's in a different file. That file can be autoloaded as well. For example, suppose that the command `c:att-chg` is in the file `att-chg.lsp` and that it's called by the `c:att-ed` command. We can specify that we want the file autoloaded as follows:

```
Command: (autoload "att-chg" '("att-chg"))
nil
```

When the `c:att-chg` command is called by `c:att-ed`, `att-chg.lsp` is automatically loaded.

Notes

- The filename must be expressed as a string, as must each command name. We *don't* put `c:` in front of the command names.
- `autoload` is not a subr; it's an application that is defined in the `acadr13.lsp` file. This file is loaded at the start of a drawing session immediately prior to the `acad.lsp` file. There is also an `autoxload` function for ADS files.
- At the time this book went to press there was a bug in `autoload`. Consult the `readme.txt` file on the enclosed floppy for the latest status of this error.

13.2 Providing Help for Commands We Write

AutoLISP provides us with the capability of supplying help for any AutoCAD command that we write. By adding help we can greatly increase the user-friendliness of our commands. Autodesk has gone so far as to integrate the help facility with Windows, but the DOS and Windows help files are pretty much the same.

Help is a Release 13 facility.

This discussion assumes that you are familiar with the standard AutoCAD or Windows help facility. Furthermore, you'll follow the explanation better if you type in the commands and create the help files as you read.

The first task that we must perform to enable the user to access help for our commands is to create a help file containing the command names and the help text for each command. Let's create a Windows help file and call it `tutorial.hlp` (create `tutorial.ahp` if you're running DOS). Each command entry in this file must contain *help directives,* which begin with a backslash and specify the entry to the help facility.

`\#` - Topic ID	A label that the help facility uses internally to locate this entry. It also serves to signal the start of a new entry.
`\$` - Title	A label that appears in the Go To directory when the user selects the appropriate keyword from the Search directory. This label is also listed in the History directory.
`\K` - Keywords	One or more strings, separated by semicolons, that the user can type to search for this topic. The help facility lists these keywords in the Search directory.

\# is the only required directive, but it's a good idea to include the other two. Following these directives we put the help text itself. In addition, there are two more help file directives:

`\E` - EOF	This end of file directive *must* be the last entry in the help file.
`\ <space>`	Any line beginning with a backslash followed by a space is taken to be a comment and ignored.

As an example we'll create help for the DRAW program, which we examined in Section 13.1.4. As we saw, the `c:draw` command calls the `choose-obj` function, which calls `draw-list`, which in turn calls `l-box`. We'll begin by creating a help file entry for `c:draw`. You may find it too lengthy to type in completely, but it shows the kind of information we might want to include in the help file.

```
\#draw
\$c:draw command
\Kdraw
The DRAW program is an on-line tutorial that pictorially exhibits AutoLISP data and code. When
you enter AutoLISP data (such as the list ((8 . "0") (62 . 2))) or an expression (such as (* 3
(+ 4 5))), it draws a pictorial representation of your input on the screen.
The pictorial representation is not only more easily understood than the normal printed represe
ntation, it is also more accurate.
The c:draw command calls the choose-obj function to determine the kind of AutoLISP object to dr
aw.
```

Notice that we do not put spaces between the directives and the text that follows. Furthermore, we don't press <enter> at the end of each line of text. When the help facility encounters a carriage return it begins a new paragraph and separates paragraphs with a blank line (see Figure 13.1).

The carriage return following "input on the screen." indicates the end of the first paragraph. A blank line will be inserted in the help text before the new paragraph that begins with "The pictorial representation . . .". We can eliminate the blank line between paragraphs by ending the first line with a backslash (e.g., we could have ended the first line of text with "input on the screen.\").

After we've created a help file, we can *register* a command with the help file to allow transparent help for that command.

`setfunhelp` str [str] [str] [str]	Associates an AutoCAD command with a help file. It returns the command name if it succeeds, `nil` if it fails.

Once a command has been registered with the help file the user can type `'help` or press the `F1` key (Windows only) during a pause in the execution of that command to bring up the appropriate help information.

`setfunhelp` for our `c:draw` command might appear as follows:

```
Command: (setfunhelp "c:draw" "tutorial.hlp" "draw")
"c:draw"
```

The first argument to `setfunhelp` is the name of the command expressed as a string; the second is another string that specifies the help file name. The *AutoLISP Reference* lists the second argument as optional, but if it's left out, the user's request for help invokes the standard AutoCAD help screen. The `.hlp` or `.ahp` extension *is* optional, however.

The third argument is also optional. In our example it tells the help facility to display text information for the Topic ID `draw` when the user asks for help on `c:draw`. The third argument could be the name of a different topic altogether. For example:

```
Command: (setfunhelp "c:run" "tutorial.hlp" "draw")
"c:run"
```

`c:run` is a minor command in the DRAW program. When the user types `'help`, we'll bring up help on the main `c:draw` command. If the third argument is missing, the first page of help text is displayed. This is typically a table of contents.

The fourth argument to `setfunhelp` is explained in conjunction with the `help` function.

If we redefine and reload our command, the help facility loses its registration of the command. Thus, we must rerun `setfunhelp` after the `defun` is loaded. One way to ensure that our command is always registered is to place the `setfunhelp` function at the end of the file in which the function is defined. In that way it is run as a script whenever the file is loaded.

Once a command has been associated with the proper file, the user can invoke its help by entering `'help` or pressing the `F1` key (Windows only) during a pause in the command. Another way to access help is to call the `help` function.

`help` [str] [str] [str]	Takes the names of a help file and a user-defined AutoCAD command and calls up help for that command. It returns the name of the help file.

```
Command: (help "tutorial.hlp" "draw")
"tutorial.hlp"
```

As with `setfunhelp`, the `.hlp` or `.ahp` extension is optional. This invocation of `help` brings up the text for the `c:draw` command in the help dialog box:

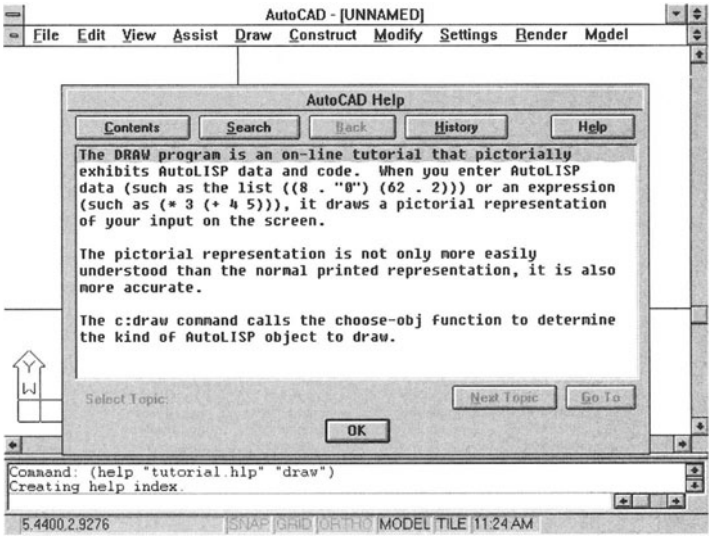


Figure 13-1

Once in help we use the standard help mechanism to move around the facility. Keep in mind, however, that this is our own help file, not AutoCAD's. Windows users can access AutoCAD's help files from within our help facility by pressing `F1`.

Since one of the arguments to the `help` function is the help file name, it is doubtful that our users will call this function explicitly to receive help. It's more likely that we would have, say, a Help button in a dialog box, then call the `help` function when that button is selected.

The definitions of the three arguments to `help` are identical to the last three arguments to `setfunhelp`. `setfunhelp` is used to create the default settings for these arguments. If the help facility is invoked via the `help` function (as opposed to `'help` or the `F1` key), `help`'s arguments override `setfunhelp`'s defaults.

To see this, let's create a help file entry for the `c:run` command. We add this entry to the help file immediately below the first entry's text; the help facility recognizes `\#` as the beginning of a new entry. We must remember to keep the `\E` at the end, however.

```
\#run
\$RUN command
\Krun
The C:RUN command executes an expression that has just been created by the C:DRAW command and
draws or prints the result on the screen.
\E
```

Now let's again issue the `setfunhelp` function we ran earlier:

```
Command: (setfunhelp "c:run" "tutorial" "draw")
"c:run"
```

When the user presses `F1` or types `'help` at a pause in the `c:run` command, the help text for the `c:draw` command is displayed. However, we can specifically request help for the `c:run` command:

```
Command: (help "tutorial" "run")
"tutorial"
```

In this case the `setfunhelp` setting is overridden and help for the `c:run` command is shown instead.

The last optional argument to either function specifies the initial state of the help window. It can be one of the following strings:

- `"help_contents"` — Displays the first topic in the help file. This is typically a table of contents.
- `"help_helponhelp"` — Displays the standard AutoCAD explanation of how to obtain help.
- `"help_partialkey"` — Displays the search dialog using the previous argument as the initial search text.
- Windows only: — A string used by the *fuCommand* argument of the WinHelp( ) function.

The `"help_partialkey"` option gets the user into the help facility but requires him to browse for the information he needs. For example, suppose we issue the following `setfunhelp` function:

```
Command: (setfunhelp "c:run" "tutorial" "draw" "help_partialkey")
"c:run"
```

When the user presses `F1` or types `'help` at a pause in the `c:run` command, the help text is not displayed. Instead, the Search dialog is brought up with `draw` as the initial search text, as shown in Figure 13-2:

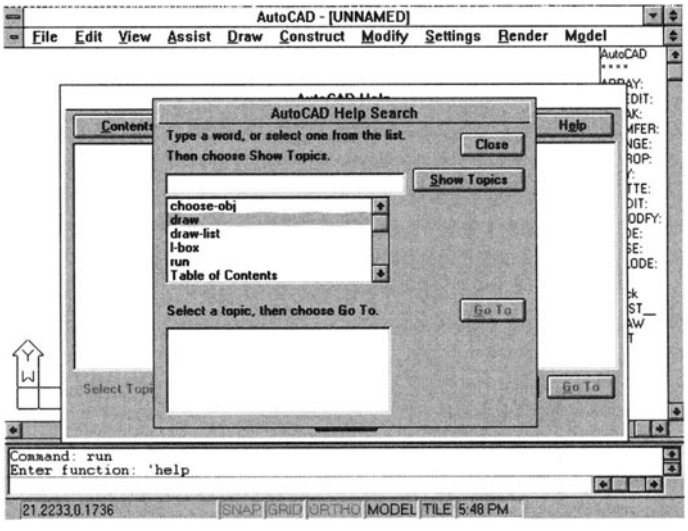


Figure 13-2

Hypertext Links

An important aspect of the help facility is our ability to integrate hypertext links in the help text. A *hypertext link* is a connection between a highlighted keyword (called a *hot spot*) in the text and help about the keyword. In AutoCAD help, words and phrases that are hot spots are printed in green; when we click on a green word, we see help about that word. Our own hot spots are shown differently but work in a similar fashion.

Let's add entries to our `tutorial.hlp` file for the `choose-obj`, `draw-list`, and `l-box` functions and place hypertext links in the help text. Add the `choose-obj` entry to the help file immediately below the `c:run` text and remember to keep the `\E` at the end.

```
\#choose-obj
\$choose-obj function
\Kchoose-obj
The choose-obj function determines what type of object its argument is. It then calls <<draw-
list>>draw-list>, <<draw-cons>>draw-cons>, or <<draw-atom>>draw-atom> to draw the object.
\#draw-list
\$draw-list function
\Kdraw-list
The draw-list function is called by the <<c:draw>>draw> command to draw a list on the user's
screen. It recursively calls <<choose-obj>>choose-obj> to draw each element and <<l-box>>l-b
ox] to draw each list box.
```

```
\#l-box
\$l-box function
\Kl-box
The l-box function draws list boxes on the screen.
\E
```

In the `draw-list` text we designate three hot spots, `<<c:draw>>`, `<<choose-obj>>`, and <<l-box>>, by putting them in double angle brackets. Following the hot spot we specify the Topic ID, which is the first directive for each entry. To the right of the Topic ID we place either a right angle bracket or a right square bracket. The Topic ID and closing bracket do *not* appear along with the help text.

The angle bracket (`draw>` in our example) designates a *topic jump*. When the user clicks on this hot spot the help text for `draw` *replaces* the original `draw-list` help text.

The square bracket (`l-box]` in our example) indicates a *pop-up* topic. When the user clicks on this hot spot, the help text for `l-box` pops up *on top* of the `draw-list` help text. When the user clicks on the Close button, the original `draw-list` help text is restored. The user can click on a hot spot in a jump window to obtain further help but can't do so in a pop-up window.

Let's run the `help` command to bring up the `draw-list` help text:

```
Command: (help "tutorial.hlp" "draw-list")
"tutorial.hlp"
```

When we do so, the help window shown in Figure 13-3 appears on our screen.

Our text is shown with the hypertext hot spots in double angle brackets. Unlike standard Windows hypertext, the user doesn't have to select the actual hot spot; clicking anywhere on its line of text is sufficient.

In our example two hot spots occur on the same line. Regardless of where on the line the user clicks, `choose-obj` becomes the Selected Topic because it appears first in the line. To make `l-box` the Selected Topic, the user must click on the Next Topic button. When she subsequently clicks on the Go To button, a window with the appropriate help text appears on the screen. To simplify matters for the user it's nice to put only one hot spot on a given line. Then she can simply double-click on the hot spot to bring up its help text.

Adding a Table of Contents

The first entry in the help file is displayed if we give `setfunhelp` only two arguments or give `help` only one argument. It is also is brought up if we supply

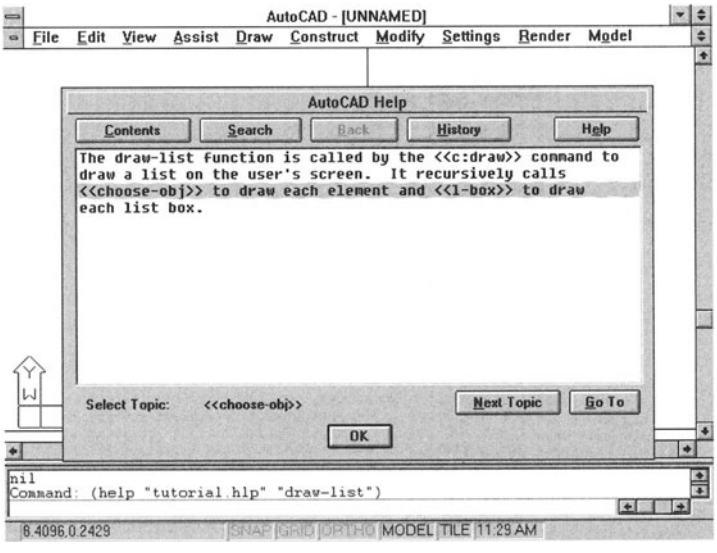


Figure 13-3

the `"help_contents"` string to either of these subrs. If we plan to build a complete help facility, it is elegant to begin the file with a table of contents (TOC). A TOC typically comprises a listing of the commands for which the user can obtain help. The command names are designated as hot spots, with hypertext links to their actual entries. For example, we might create the following initial help file entry:

```
\#toc
\$Table of Contents
\Kcommands; functions; TOC, table of contents
<<c:draw>>draw>\
<<choose-obj>>obj>\
<<draw-list>>list>\
<<l-box>>lbox>\
<<c:run>>run>\
```

The backslash at the end of each line eliminates the blank line that would otherwise appear between each hot spot in the help file; it keeps the lines contiguous. If we have many items in our TOC, we can put two or three on each line. However, the user must then click on the Next Topic button to sequence among the choices.

The following function calls show several ways in which the table of contents would get displayed. In all cases the following appears on the user's screen:

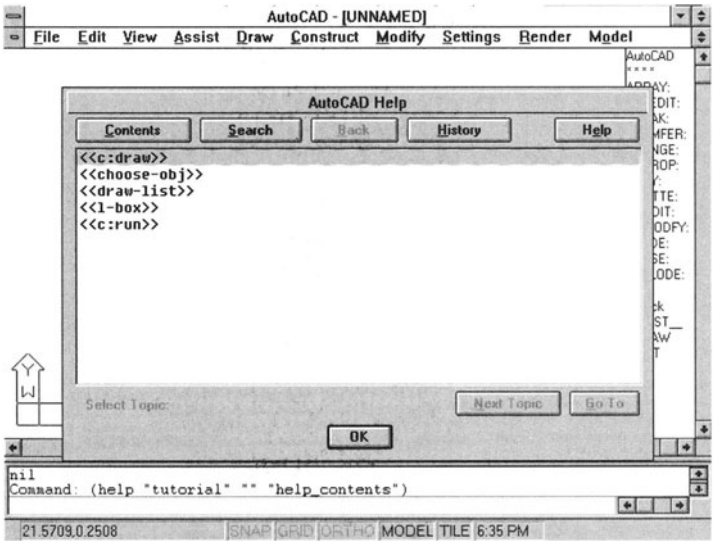


Figure 13-4

```
Command: (setfunhelp "c:draw" "tutorial.hlp")
"c:draw"
```

This call to `setfunhelp` tells the help facility to look in the file `tutorial.hlp` when the user requests help for `c:draw`. However, since the third argument to `setfunhelp` is omitted, the first entry in the file, which is the TOC, is displayed when the user asks for help. The following call to `setfunhelp` causes the TOC to be displayed as well:

```
Command: (setfunhelp "c:draw" "tutorial.hlp" "" "help_contents")
"c:draw"
```

We use the null string as a placeholder because a third argument must be present for us to supply `help_contents` as the fourth argument. Since the previous `setfunhelp` call achieves the same result, the `help_contents` argument isn't really necessary. However, it does clarify our intentions.

Having issued `setfunhelp`, suppose we now run the `help` function, as follows:

```
Command: (help "tutorial" "draw-list")
"tutorial"
```

In this case the help text for the `draw-list` function is displayed on the screen instead of the TOC, because the arguments to the `help` call override the `setfunhelp` defaults.

We can have the `help` function display the TOC in one of the following ways:

```
Command: (help "tutorial")
"tutorial"
```

or

```
Command: (help "tutorial" "" "help_contents")
"tutorial"
```

Again, the former requires less typing, but the latter shows our intentions more explicitly.

♦ ♦ ♦

The help facility has a couple of restrictions. If our command doesn't pause (e.g., for a `getstring` or `entsel`), the user can't request help. In fact, `(getstring t)` won't permit a help request either. Another limitation is that we can only run `setfunhelp` on AutoCAD commands that we write (i.e., ones that begin with `c:`). Our AutoLISP functions aren't processed by AutoCAD's command processor, so we can't associate help with them.

We can overcome these restrictions by writing an AutoCAD command whose sole purpose is to allow the user access to our help file:

```
(defun c:gethelp (/ cmd)
  (setq cmd (getstring "\nEnter command or function you need help for: "))
  (help "tutorial" cmd)
  (princ))
```

The `c:gethelp` command asks the user for the name of the command for which he wants help, then explicitly calls `help` for that command. Once in help he can search for the topic of interest, be it a command or a function. In other words, we can put function names in a table of contents (as we did with `choose-obj`, `draw-list` and `l-box`) and the user will be able to obtain help for them.

Note: Consult the `readme.txt` file on the enclosed floppy for the latest status of help facility bugs.

13.3 Understanding Other Programmers' Code

One of the least enjoyable tasks that can be foisted upon us is to inherit files of indecipherable code that were written by an antisocial, part-time hacker who has long since departed the company, and having to bring that code up to date for the next release.

When this situation occurs, our initial goal should be to reorganize the code into a readable form, without necessarily changing any of the logic. Being able to read the code is a major first step.

Learning to trace through code is a skill that can only come from practice. This section is merely intended to get us going in the right direction by examining suspect code and seeing how it might be cleaned up. Keep in mind that if an expression is complex we can often enter its components at the keyboard, starting with the innermost parentheses, to learn what it is doing.

In my travels I have acquired the following delicious feast of stylistically improper code. In some ways this is a well thought out and sophisticated program. But the coder is obviously unaware of the many coding techniques, style points, and tricks that are by now quite familiar to us. On the succeeding pages the original code is presented along with comments regarding its structure and organization. Afterward a possible revision to the code is displayed.

It's uncertain what this code actually does because we don't have access to the data files that it loads. This hampers our attempts to completely clean it up. However, in our examination of its structure, the code's intent is secondary.

The program is organized in a logical fashion. It consists of 10 functions that are called by the following two commands situated at the end of the file. Having them at the end is OK, but it's nicer to place them up front so that the reader can immediately see the flow of the code.

```
(defun c:replace ()
   (partnos)
   (filegrab)
   (varset)
   (dimcheck)
   (placstr)
   (placnum)
   (placpat)
   (titleblk)
   (callscr)
)
```

```
(defun c:repete ()
   (filegrab)
   (varset)
   (dimcheck)
   (placstr)
   (placnum)
   (placpat)
   (titleblk)
   (callscr)
)
```

As we can see, these two commands are virtually identical; the only difference is the call to `partnos` in the `c:replace` command. Therefore, we can collapse the two commands into one with the following modification to `c:replace`:

```
(defun c:replace ()
  (partnos)
  (c:repete))
```

All `c:replace` has to do is call the `partnos` function, then run the `c:repete` command.

Notes

- The entire program is fairly long, so we'll just examine the `partnos`, `filegrab`, and `varset` functions.

- There are no comments in the portion of the original program that is presented here; all comments are mine. Unfortunately, this omission makes the example more authentic!

- The coder does a nice job of showing where the special operators end by putting close parentheses under the matching opens. However, as we shall see in the revised version, proper indentations do more to clarify the code.

- All of the functions are called from the main command, which causes a problem with variables. For example, the central purpose of the `filegrab` function is to open a file. But the file isn't used in `filegrab`; it's used in the following function, `varset`. This means that `zz`, which names the file descriptor, must be a global variable. If `filegrab` called `varset`, it could pass the file descriptor as an argument and the variable `zz` could be local.

  If `zz` is global, we should give it a more obscure name. Also, a more informative one. The same can be said of the variable `i`, which suffers from the same problems. Their common names can readily cause conflicts.

Here's the original code with my comments:

```
(vmon)                                                  ;VMON is now obsolete
(defun partnos ()                                       ;Most variables should be declared local
  (setq a (list a))                                     ;Since A is a list let's rename it LIS
                                                        ;This code creates the list (nil), not
                                                        ;  what we want. Since local A is nil by
                                                        ;  default, this SETQ isn't needed.

                                                        ;Here's a perfect place for the Y-NP
                                                        ;  function from Chapter 8 lab
(prin1 "Enter <y> for Yes, <n> to Enter Part Numbers directly.")
(setq dec1 (getstring "\nMake print of all liners just entered in basic program? "))
  (if (or (= dec1 "y")(= dec1 "Y"))                     ;These could be tested at once with
                                                        ;  STRCASE. However, the INITGET in Y-NP
                                                        ;  eliminates the need for this test.
    (progn
       (setq pnfil "rehfil.txt")                        ;Do multiple assignments in 1 SETQ
       (setq pno "start")                               ;Use T to mean non-nil, not "start"
       (setq rhfil (open pnfil "r"))                    ;Say (open "rehfil.txt" "r") and
                                                        ;  eliminate the PNFIL variable
    (while (/= pno nil)                                 ;Merge these two lines into one
         (setq pno (read-line rhfil))                   ;  and indent a little less here
         (setq bz (cons pno a))                         ;Say (setq a (cons pno a)) and
                                                        ;  eliminate the BZ variable
         (setq a bz)                                    ;Use a better name than A
      )
    )                                                   ;Good use of parens
  )
  ;;If the user typed something other than "Y" or "N" this code doesn't
  ;;  check for it. If the user is restricted to "Y" or "N" (i.e., if we
  ;;  use Y-NP) then the following test is superfluous.
  (if (or (= dec1 "n")(= dec1 "N"))
    (progn                                              ;PROGN isn't needed
      (while (/= pno "none")
        (setq pno (getstring "\nPart Number: "))
        (setq bz (cons pno a))                          ;Again BZ isn't needed
        (setq a bz)
      )
    )
  )
  (setq a (reverse a))
  ;;The variable I isn't used here, so it should be set in FILEGRAB.
  ;;If we delete the unneeded (setq a (list a)) (above) then I should be
  ;;  initialized to 0 so that the following NTH gets the first element.
  (setq i 1)
     (setq bz nil)                                      ;Why bother? BZ isn't used again
     )
```

```
(defun filegrab ()
     (setq fil (nth i a))                              ;Watch the indents.
                                                       ;Put following COMMAND on its own line.
   (if (or (= fil "none")(= fil nil))(command "script" "getout"))
   (if (or (/= fil "none")(/= fil nil))                ;Don't need both IFs.
     (progn
         (setq zz (open fil "r"))                      ;If 1 SETQ, we don't need PROGN
          (setq i (1+ i))
     )
   )
)                                                      ;Skip lines between DEFUNs
;;;The original VARSET function has 18 SETQs; we'll just show 3.
(defun varset (/ ul1 upl1 mw7d1 t1 bead1 diag1 hbd1 z321 rad1 b31 cor1 hdsd1 hssd1 hbr1 leg1
    o331 sbhsr1 upc1)
    (setq ul1 (read-line zz) ul2 (atof ul1))           ;Don't need UL1 etc.
    (setq upl1 (read-line zz) upl2 (atof upl1))        ;  so don't need locals
    (setq mw7d1 (read-line zz) mw7d2 (atof mw7d1))     ;Put all in 1 SETQ
            .
            .
            .
))
```

Following is the revision to the preceding functions. The revised code should be completely commented, but several design flaws in the original code prevent us from fully understanding the coder's intent. For example, the variable `i` is set up as an index into the list, but no looping mechanism has been created to use it. Still, our revisions make the code readable; in the original code, errors such as this one are completely obscured.

```
(defun partnos (/ lis rhfil pno)
  (if (y-np "\nMake print of all liners just entered in basic program? ")
      (progn                                           ;Yes,
        (setq rhfil (open "rehfil.txt" "r"))           ;Open file
        (while (setq pno (read-line rhfil))            ;If more data in file,
          (setq lis (cons pno lis))))                  ;  cons data onto list
      (while (/= pno "none")                           ;No, until user enters "none",
        (setq pno (getstring "\nPart Number: ")        ;  get next string
              lis (cons pno lis)))                     ;  and cons onto list
  )
  (setq lis (reverse lis)))                            ;Reverse list when done

(defun filegrab (/ i fil)
  (setq i   0
        fil (nth i lis))                               ;Get first list element
  (if (or (= fil "none")(null fil))                    ;If done
      (command "script" "getout")                      ;Quit
      (setq zz (open fil "r")                          ;Oops, this is going to bomb!
            i  (1+ i))
  )
)
```

```
(defun varset ()
  (setq ul2   (atof (read-line zz))
        upl2  (atof (read-line zz))
        mw7d2 (atof (read-line zz))
              .
              .
              .
  ))

;;;Here's an alternative approach to VARSET when there are many variables to set.
(defun varset2 ()
  (setq l '(ul2 upl2 mw7d2 t2 bead2 diag2 hbd2 z322 rad2 b32 cor2
            hdsd2 hssd2 hbr2 leg2 o332 sbhsr2 upc2))
  (foreach elt l
    (set elt (atof (read-line zz)))))
```

`varset2` is a clever algorithm for assigning each line of a file to a variable. It first makes a list of all the symbols whose values it needs to set. `foreach` then sequences through the list and assigns values to each with the `set` function. We need to use `set` (rather than `setq`) here because we don't want to bind `elt`, we want to bind the *value* of `elt` on each pass. `atof` is used in this function because each line of the file is the ASCII representation of a real number and it must be converted.

This concludes our discussion of programming style. I hope that you'll incorporate these programming techniques in your daily work. They really are what differentiate coders from accomplished programmers. When we make the effort to write elegant programs, we usually find that our programs are more efficient and error-free as well.

VI

Introduction to the Computer

Many AutoCAD users find learning AutoLISP difficult because a knowledge of programming basics is often assumed and they've never had formal computer training. Well, help is here!

Chapter 14 explains many computer concepts that will aid your understanding of the AutoLISP language. Depending on your background and experience, you may know some of this information or you may know it all. If you're new to programming, however, you might want to begin your study of AutoLISP by reading this chapter. It's divided into three sections:

14.1 covers the hardware components that constitute a computer. This is purely background information and not required for programming in AutoLISP.

14.2 explains how data are stored in the computer, Boolean algebra, ASCII files, and memory management. If you're just beginning your study of AutoLISP, you'll find the descriptions of AND, OR, and ASCII representation

invaluable. The memory management section will appeal to more-experienced users, who will also find the explanation of Boolean algebra germane to some of the more advanced AutoLISP functions.

Section 14.3 contains *essential* information for anyone who wants to use AutoLISP. This section explains what computer programs are, what they do, how we create them, and how we run them. Important programming concepts, such as looping and recursion, are also discussed. We introduce a text editor (which is on the enclosed floppy) that you might find more powerful for AutoLISP than the one you're currently using. Finally, AutoCAD's `acad.pgp` file is examined. If any of these concepts are new to you, please take the time to read Section 14.3; the information it contains is assumed throughout the rest of the book.

14

Computer and Programming Basics

Topics Covered in This Chapter

- Computer components
- Data representation and storage
- Writing and running programs

Goals for This Chapter

The primary intent of this chapter is to provide background information for AutoCAD users who have never been formally exposed to computer fundamentals. Our goals are to:

- Learn the hardware components of a computer. Study their capabilities, limitations, and relationships to each other. Understanding a computer's organization enables us to better utilize its resources.
- Learn how data are represented and stored in the computer. This is a bit-level introduction to how a computer "thinks." Certain frequently used terms are explained, such as *bit, byte, word, Kb, ASCII,* and *virtual memory.*
- Learn software fundamentals: how programs are written and run; why we write them; the difference between compilation and interpretation; the edit-compile-debug cycle; looping and recursion. *These concepts are extremely important, and their knowledge is assumed throughout the book.* Even if you are an experienced programmer, scan Section 14.3 to ensure that you understand all of these concepts.
- Learn the AHED editor. We must know a text editor to write programs, and AHED has some features that make it particularly attractive for AutoLISP programmers. Even if you are already using an editor, it is worthwhile spending a few minutes studying AHED to see if it might better fit your needs.
- Discover the `acad.pgp` file, which is our key for running DOS-level programs from within AutoCAD.

Introduction

As AutoCAD users we sit at a computer for several hours every day. To have a common frame of reference when learning what AutoLISP can do for us, we should have at least a cursory knowledge of the various hardware components of a computer system. There are also a number of concepts and terms that are useful to know when talking about computers. An understanding of these will facilitate your study of AutoLISP.

It is easy for those of us who have been working with computers for many years to forget the questions we asked when we first began to program. We tend to assume that new users are aware of concepts that they really have no reason to know. This chapter is intended to fill in those gaps in your knowledge and explain terms that programmers use every day. Not all computer terms are covered in this chapter, only those that are particularly interesting or have relevance to the use of AutoLISP.

Some concepts explained here will be new to you; others you might be quite familiar with. That's fine; take just what you need from this chapter and skim or skip the rest. The facts that you know will probably be new to someone else.

14.1 Computer Components

When we talk about the hardware that constitutes a computer, we usually divide the components into four categories: the CPU, main memory, auxiliary storage devices, and input/output devices.

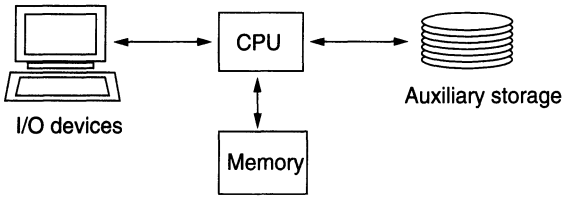


Figure 14-1

Let's examine what each of these components does.

Central Processing Unit

The central processing unit (CPU) is the "brain" of the computer. When we type a command to DOS, run a program in memory, or click with our mouse on a drawing, the CPU interprets the commands it's been given. It then performs the following kinds of tasks:

- It acquires and directs the resources that it needs, such as a printer, a disk, or the math coprocessor.
- It runs DOS commands, editor commands, and AutoLISP programs.
- It sends data to the terminal screen.

As you are probably aware, to run AutoCAD on a PC we must have a math coprocessor. This is a chip that aids the main processor by performing calculations at lightning-fast speed. Although the coprocessor runs very quickly, it is not endowed with the same intelligence as the CPU. It merely performs the mathematics component of a small set of operations, such as:

- Calculating distances and lengths
- Offsetting and mirroring lines
- Deciding how to fit text

Main Memory

Main memory is the computer's "scratch pad" area. In order for the computer to perform work on data, that data (as well as the program that is operating on that data) must be in main memory. In general, memory is large enough to hold several programs at once. However, under DOS, only one program can be executed at a given time.

Here's an analogy: When we bring our car in to be serviced, ten other cars may be waiting for servicing along with ours. The garage might have only two bays, meaning that it can repair at most two cars at a time. The other eight cars must wait outside in the parking lot. Furthermore, if the service station has only one mechanic, then it can repair only one car at a time, despite the fact that two cars are in the garage.

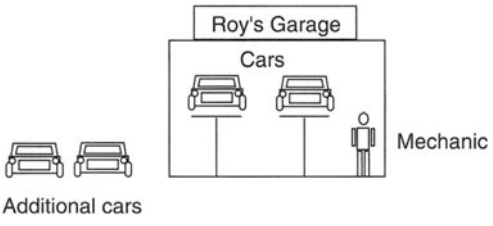


Figure 14-2

There is only one CPU and one main memory on a PC, just as there is only one mechanic and one garage. As there might be two cars in bays waiting to be serviced, so too there can be more than one program (for example, AutoCAD and your text editor) in memory waiting to run (the others reside on the hard disk). But only one car or one program can be serviced at a given time.

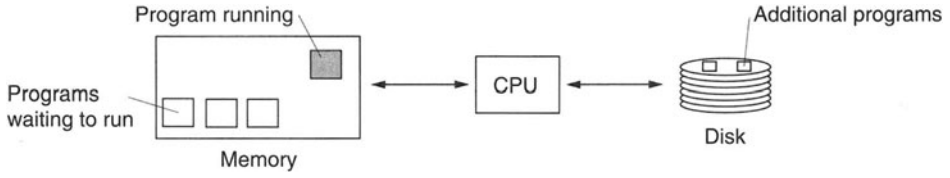


Figure 14-3

Some larger computers can be *timeshared,* or used by several programmers at the same time, giving the appearance of running more than one program at a time. This is also known as *multiprogramming.*

Multiprogramming is akin to having the single mechanic run back and forth between the bays. Each car is being serviced, but not exclusively. While the first car is running a test the mechanic can work on the second car. In this manner the mechanic has no waiting time, and the job is completed on both cars faster than if the mechanic had worked on them sequentially. Timesharing on computers is efficient in a similar manner, especially because a computer doesn't need coffee breaks! While one user's program is waiting for the printer another program can utilize the CPU. However, the CPU is really processing only one job at any given moment.

Some computers have two or more processors, which allows them to operate on two or more pieces of data concurrently . . . rather like having two mechanics in the garage. This configuration is called *multiprocessing.*

Memory consists of both *read-only memory* (ROM), which contains instructions for the computer, and *random-access memory* (RAM), which holds our

programs and data. RAM is *volatile,* which means that when we shut off the computer (or it "goes down" accidentally), whatever we have loaded into main memory is destroyed. This is why, when we are editing code or modifying our drawings, we should frequently save our work onto the hard disk. The hard disk retains its information even after the computer has been shut off. The hard disk is one example of an auxiliary storage device.

Auxiliary Storage Devices

A wide variety of auxiliary storage devices exists, including the following:

Hard disks

Our hard disk (or disks) are usually the `c:` and `d:` drives. They hold massive amounts of information, possibly hundreds of millions of bytes (called *megabytes* or *mb*), compared with 1 to 32 mb of main memory on a typical PC.

The disk is a *random* access medium; as with compact disks, we can access the information in any order we choose. Because it holds so much data and the data are so easily accessed, we typically store our files, including most of our current drawings, on the hard disk.

The CPU can access data in memory extremely quickly as compared with disk access time, because bringing in data from the disk is a relatively slow process. Compare how fast we can access portions of a drawing that is already in memory with the time needed to read the same information from a drawing that is on disk. Speed aside, for the CPU to run a program or perform any kind of work on data, that data must first be brought into memory. Thus, the disk is used for storage purposes only.

Some computers have removable hard disks, but they're still less portable than floppy diskettes.

Floppy diskettes

Diskettes or "floppies" are convenient because they are small, light, portable, and random access. The 5¼" floppies hold either 360 kilobytes or 1.2 megabytes. The smaller 3½" diskettes, which hold 1.2 to 1.4 mb, are made of hard plastic but are still referred to as "floppies." Diskettes typically occupy the `a:` and `b:` drives on DOS machines.

Floppies are a convenient medium for transferring code or data from one machine to another. They're also popular for backing up data because they're small and easily stored. Because hard disks occasionally break or have their data scrambled, it is very important that we periodically save our information onto a backup source.

There is a noticeable difference in access time between hard disk and floppy. Because the hard disk holds much more data than a floppy and the access time is significantly less, we use a hard disk to hold data we access frequently, and floppies to hold data that we want to transport or store long-term.

CD-ROM

A CD-ROM is a ***C***ompact ***D***isk with ***R***ead-***O***nly ***M***emory. Its main advantage is its ability to store and quickly access large amounts of data. Presently, however, writable CDs aren't commonly available. Once they are, they'll probably replace floppies for backups and large application storage.

Magnetic tapes

We can store tremendous amounts of information very cheaply on magnetic tapes. Like the tape in our cassette decks, this is a *sequential* access medium, meaning that the information must be accessed in the same order that it was stored. Its attractive price coupled with this inconvenient method of access makes magnetic tape an excellent medium for backing up huge amounts of data or storing large databases.

Input/Output Devices

A multitude of Input/Output (I/O) devices exists, and you're probably familiar with most of the ones that are commonly used with AutoCAD.

Consoles

Console is a term frequently used to identify our workstations. The console comprises the keyboard; a digitizer, mouse, or other pointing device for data input; and the terminal screen for output.

Printers

When we write AutoLISP programs we frequently want hard copy listings of our code, so it's important to have access to a printer. Printers can also "plot" drawings much faster than plotters can.

Plotters

As you know, we can get beautiful, multicolored renditions of our drawings on plotters. However, we don't plot code; we plot drawings that are created and modified by programs we write. Thus, plotters do not play a part in the story that this book tells.

14.2 Data Representation and Storage

Data are represented in a few different ways on a computer. People sometimes expect data to be stored in a readable format, but programming languages often require a more concise data representation that is undecipherable by humans. This section explains how data are represented and stored inside the computer.

Much of the information in this section is important for understanding a small number of AutoLISP functions. At the start of each topic we'll list the functions to which that topic pertains; you can come back to the topic when you're ready to learn those particular AutoLISP functions.

Computers have evolved tremendously in the years since they were huge dinosaurs that filled a gigantic room. Memory, which used to be built of thousands of small metal donuts called *cores* (you may have heard the term *core memory*), now consists of tiny chips that are wired onto a board in our PC.

One thing that has not changed, however, is the essential yes-no nature of the computer. At the very lowest level all information is stored as a series of *bits*. A bit is very analogous to a lightbulb; it's either on or off. If it's turned on, we say that the bit is set to 1. If it's off, it is set to 0. Through judicious setting of these bits, the computer stores and conveys information.

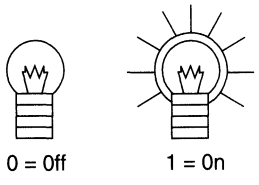


Figure 14-4

A *byte* is simply a collection of bits. The number of bits that constitute a byte differs on some types of machines. On a PC there are eight bits in a byte. Collecting bits into larger units makes handling information easier. For example, each character we enter at the Command: prompt is represented in one byte of memory.

Through the setting of bits, it is very easy to ask the computer yes-no type questions. If the answer is yes, then a bit is turned on; otherwise it is turned off. In other words, at its lowest level, the computer is a binary machine.

14.2.1 Binary Representation of Numbers

This topic is important for the bit manipulation operators.

When doing everyday mathematics, we work in base 10, or *decimal.* Humanity has done so since the earliest times, when people used their fingers to count. Computers count using bits, which are either off or on, 0 or 1. Because each bit only represents two choices, this is a base 2 or *binary* number system.

In decimal, the largest digit is 9. If we add 1 to it, there must be a carry into the second column, which gives us 10. In binary, the largest digit is 1, and when we add 1 to it the same thing happens . . . we get 10. However, this is "10" in base 2, which is equivalent to 2 in base 10. The following table shows the progression of the first 16 numbers in decimal and binary:

Decimal	*Binary*	*Decimal*	*Binary*
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Table 14-1

The smallest number that can be represented in four bits is 0000, which is the same as 0 in base 10. If we add 1 we get 0001, or 1 in base 10.

Since there is no number 2 in binary, when 1 is added again we are at the limit of data that the first bit can hold. The information is therefore carried into the second column, giving 0010. This is equivalent to 2 in base 10. If we again add 1, we get 0011, the binary representation of 3.

Sometimes it is unclear whether a number is binary or decimal. When clarification is required, the base of the number is specified via a subscript, as in the following equation:

$$11_2 = 3_{10}$$

Don't confuse the subscript 5_2 (which means 5 in base 2) with superscript 5^2 (which means 5 squared).

When we have the number 3 (0011_2), the first two digits are full, so if we add 1 again there is a carry into the third column, giving 0100, the representation of 4. Notice an interesting relationship: In decimal, 10^2 is 100 and in binary, 2^2 is 100. This is not a coincidence. Going a step further, in decimal, 10^3 is 1000, and in binary 2^3 is 8, which is also represented as 1000. Relationships stay the same across number bases; only the representation of the numbers differs.

A pattern is repeated. 1 is carried into a new column according to the power of 2: 2^2 is 4, so 4 is represented as 1 in the third bit position. 2^3 is 8, so 8 is represented as 1 in the fourth bit position, and so on.

In Table 14-1, the right-hand column is an exact repeat of the left-hand column except that the 8 bit is set. Aside from that bit, the representation of 11 is the same as the representation of 3. This makes sense: 3 + 8 = 11.

14.2.2 Bits, Words, and Number Representation

This topic is useful for understanding how numbers are represented in the machine and learning such terms as *kb* and *mb*.

A collection of bits used to represent an integer is called a *word.* The number of bits in a word is dictated by the type of the machine. Many PCs have 16-bit words, each of which contains two bytes. Some PCs, and other machines such as Sun workstations, have 32-bit words that can store much larger numbers.

To simplify the discussion of computer words, we'll examine a make-believe computer that has 8-bit words. The concepts are the same, but the smaller numbers are easier to work with. Table 14-2 shows the decimal equivalent of various numeric possibilities given an 8-bit word:

Binary	*Decimal*	*Power of 2*	*Binary*	*Decimal*
00000001	1	0		
00000010	2	1	00000011	3
00000100	4	2	00000111	7
00001000	8	3	00001111	15
00010000	16	4	00011111	31
00100000	32	5	00111111	63
01000000	64	6	01111111	127
10000000	28	7	11111111	255

Table 14-2

The left side of this chart shows that a solitary 1 in any bit position is equivalent to 2 raised to the power indicated by that position. We begin counting bit positions from 0; $2^0 = 1$ (in any number base, a number raised to the 0 power yields 1).

In various places the *AutoLISP Reference* talks about *bits* and *bit values*. *Bit* 0 has a *bit value* of 1; *bit* 1 has a *bit value* of 2; *bit* 2 has a *bit value* of 4, and so forth. When discussing this concept, *Essential AutoLISP* will always refer to the *bit values,* which are the numbers we must supply our programs.

For any given amount of bits, the largest number is represented by 1s in all the positions. This is shown in the right side of Table 14-2. For three bits, the largest number is 111 ($= 7_{10}$). Adding 1 causes a carry, and 1000 ($2^3 = 8_{10}$) is returned. Therefore, the largest number that can be expressed in three bits is $2^3 - 1$ ($= 8 - 1$). For our 8-bit word, the largest number is $2^8 - 1$ ($= 256 - 1$), or 255. In fact, for any word of N bits, the largest number that *could possibly* be expressed is $2^N - 1$.

A *caveat* is that negative numbers are also expressed in a computer word. To accommodate them, we designate the first bit in a word to be the *sign bit.* If the sign bit is 0, then the number is positive; if it is 1, then the number is negative. This means that any number with 1 in the first bit is a negative number.

An implication of having that first bit be a sign bit is that the largest positive integer that can be expressed in an 8-bit word is really 01111111. This is 127_{10}, or $2^7 - 1$. For a word of N bits, the largest positive number that can be expressed is actually $2^{N-1} - 1$. For a PC with 16-bit words, the largest positive number that can be expressed is $2^{15} - 1$, or 32767.

Negative numbers are expressed in an opposite manner from positive numbers. Table 14-3 shows the negative numbers that can be expressed in four bits. An 8-bit word is shown to give a feel for the larger word size.

Decimal	*Binary*	*Decimal*	*Binary*
–1	11111111	–9	11110111
–2	11111110	–10	11110110
–3	11111101	–11	11110101
–4	11111100	–12	11110100
–5	11111011	–13	11110011
–6	11111010	–14	11110010
–7	11111001	–15	11110001
–8	11111000	–16	11110000

Table 14-3

Because the first bit of a word is the sign bit, the largest negative number that can be expressed in a word of, say, three bits, is 100, which is –4 (-2^2). A 4-bit word can hold the negative number 1000, which is –8 (-2^3). Given a word of N bits, the largest negative number that can be expressed is -2^{N-1}. For a 16-bit word, the largest negative number is -2^{15}, which is –32768.

Here's one more interesting observation about binary notation: The powers of two are 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. 2^{10} is 1024, which is just over 1000. We often talk about a *kilobyte* (or simply *kb*) being 1000 bytes. But 1kb is actually 1024 bytes (or 512 words). 1 *megabyte* (or 1 mb) is over a million bytes and is equal to 1024kb. These relationships are shown in Figure 14-5.

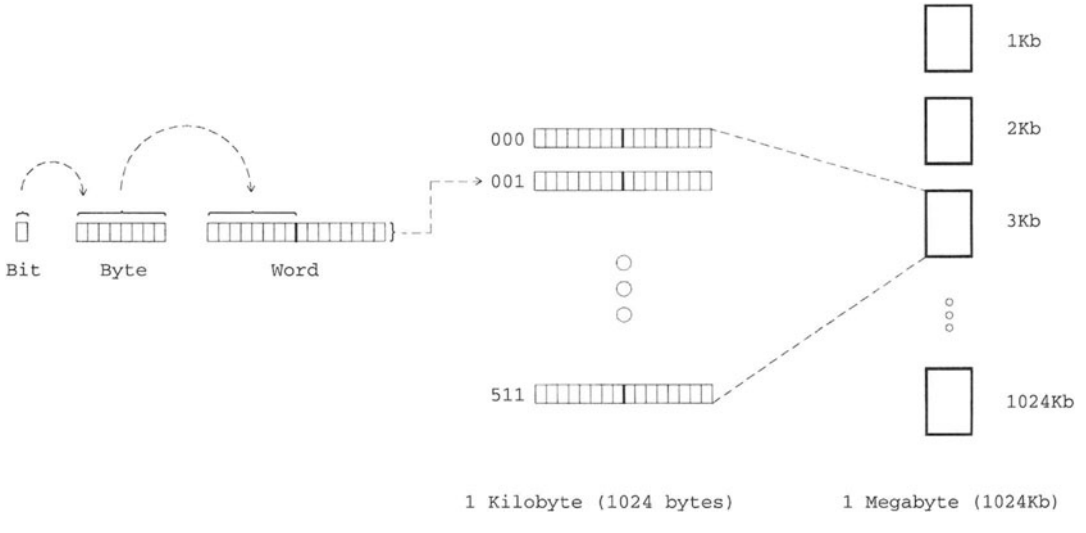


Figure 14-5

14.2.3 Boolean Algebra: Logical AND, IOR, XOR, and NOT

We must understand the concepts of AND, IOR, and NOT to program effectively in AutoLISP.

In the mid-19th century a mathematician named George Boole invented a new method of handling numbers in a logical, rather than quantitative method. This new system, which was named *Boolean algebra* after its inventor, has proven to be very useful in many fields, including logic, electrical engineering, and computer science.

In Boolean algebra one or more conditions are analyzed or compared, and True/False results are returned based on those tests. Many different Boolean comparisons can be performed, but typically only four are used for AutoLISP.

AND AND compares two or more conditions. If *all* conditions are True, then AND returns True. If *any* condition is False, then AND returns False.

In AutoLISP we might say, "Examine all of the circles in my drawing. If a circle is red and it's on Layer 1 and its radius is greater than 1, increase its radius by .3." In other words, if the circle isn't red or if it's not on Layer 1 or if its radius is too small, the program should ignore that circle.

IOR ***I***nclusive ***OR*** compares two or more conditions. If *any* condition is True, then IOR returns True. If *all* conditions are False, then IOR returns False. IOR is usually abbreviated OR, as in the case of the AutoLISP `or` function, which performs an IOR operation.

In AutoLISP, when we want to be certain that the user selected a line, we run a test that asks, "Did the user click on nothing or select an entity other than a line?" If either one of these conditions is met, we reprompt for input.

XOR ***E***xclusive ***OR*** compares two or more conditions. If *only one* condition is True, then XOR returns True. If more than one condition is True or all conditions are False, then XOR returns False.

XOR isn't used in AutoLISP as frequently as AND and IOR. However, a situation could arise in which we want to perform an operation on an entity if it is one of the following:

a circle with a radius of .5 *or*
a circle on Layer 1
but *not* a circle that is both on Layer 1 *and* has a radius of .5.

NOT NOT usually accepts only one argument and reverses it. Instead of searching for all circles whose radius is .8, we might want to perform an operation on all circles whose radius is *not* .8.

The remainder of this section is important for the bit manipulation operators. If you are new to programming, I suggest that you return to this section at a later time.

We can use Boolean algebra to perform operations on the individual bits in a word. The following table shows the results of running AND, IOR, and XOR on combinations of two bits:

Bits	*AND*	*IOR*	*XOR*
0 0	0	0	0
0 1	0	1	1
1 0	0	1	1
1 1	1	1	0

Table 14-4

AND only returns 1 if both bits are 1; IOR returns 1 unless all bits are 0; XOR returns 1 if either but not both bits are 1; NOT returns 1 when given 0 and returns 0 when given 1.

Boolean operations can be performed on numbers that are larger than 1 and 0. This is accomplished by breaking down the numbers into their 1 and 0 components. Let's examine how AND, IOR, and XOR operate on the numbers 18 (10010_2) and 6 (110_2). Note that the result is converted to decimal, but there isn't a normal arithmetic relationship between it and the original numbers.

In each example, the corresponding bits of each number are compared. It is permissible to have as many leading 0s (or 1s for a negative number) as needed. Small integers in a 16-bit word have many leading 0s or 1s.

```
AND   18     10010
       6     00110
             00010 = 2₁₀
```

For AND, the resultant bit in a given position is 1 if and only if both bits in that position are 1. Only the 2-bit in each number is 1, so the result is 2.

```
IOR   18     10010
       6     00110
             10110 = 22₁₀
```

For IOR, the resultant bit in a given position is 1 if either bit in that position is 1. This is true of the 16-bit, 4-bit, and 2-bit. Adding these returns 22.

```
XOR   18     10010
       6     00110
             10100 = 20₁₀
```

For XOR, the resultant bit in a given position is 1 if either but not both bits in a given position is 1. This is true of the 16-bit and 4-bit. Therefore, the result is their sum, 20.

Finally, NOT simply flips the bits in its single argument. A positive integer becomes negative, but the result does not have the same magnitude as the original argument:

NOT 18_{10} = 00010010_2 yields 11101101_2 = -19_{10}

NOT -3_{10} = 11111101_2 yields 00000010_2 = 2_{10}

When the bits are flipped, why don't we simply end up with the negative of the original number?

The reason is that 0 (00000000_2) is a positive number; in fact, it is the smallest positive integer. The smallest negative integer, on the other hand, is –1 (11111111_2). These two numbers complement each other; they are the binary reverse. This relationship is propagated throughout the integers, so that the NOT of any positive number is a negative number with a magnitude one greater. This is called a *one's complement* representation.

Earlier it was shown why the largest positive number on a 16-bit machine is 32767. Now you should understand why the largest negative number is –32768.

14.2.4 Storing Data and Code

This topic is background for the general understanding of data storage in computers. It's important to understand ASCII code for the conversion and I/O functions.

Data and code are stored in memory and on the hard disk in several disparate ways. Knowing these increases our understanding of why computers behave as they do.

Heaps, Queues, and Stacks

Heaps, queues, and stacks are three of the more common ways in which we temporarily store data in memory.

A *heap* is a collection of data in no apparent order. When we buy a bag of peanuts, the nuts are in a heap; it doesn't matter what order they were put in the bag, nor do we care about the order in which we remove them.

A portion of the memory available for our use is kept in a heap (called LISPHEAP in early versions of AutoLISP). When we need to create symbols or functions, AutoLISP allocates memory from that heap.

When we line up at a movie theater, we are in a *queue.* The salient feature of a queue is that the first person who gets on the queue will be the first person who enters the theater.

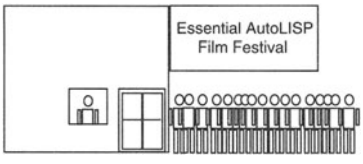


Figure 14-6

If several computers are connected to the same printer, whoever requests something to be printed first will be first on the print queue; his job will be printed before all of the other requests. For this reason, a queue is often referred to as a *first in, first out* method of storing data.

For a real-life example of a *stack,* consider our household dishes. Whenever we wash a dish we place the clean dish on top of dishes that are already stacked in

the cupboard. When we go to use a dish, we always select the dish at the very top, that is, the one that we most recently washed. Thus, stacks are considered *last in, first out* storage methods. We *push* data onto the top of a stack, then *pop* the top of the stack to access this information.

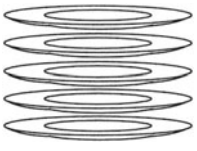


Figure 14-7

When programming, it is sometimes useful to keep track of what the user is currently doing and forget for a time what she was doing previously. In this case it is convenient for us to organize our data in stacks. When the user completes her current task, we erase the information about it that's on the top of the stack and return to the information that describes what she was doing immediately before.

When we call functions, AutoLISP stores arguments and partial results on a stack. This stack was called LISPSTACK in early versions of AutoLISP.

ASCII Text Files

Information is stored on a disk in files. Files that contain our AutoLISP programs are merely the textual representation of information. However, because a computer cannot yet understand English (or Japanese for that matter), we have to store this text information in zeros and ones. Many years ago a code was invented whereby we can represent many letters, numbers, and special characters numerically. This code is called *ASCII,* which stands for ***A***merican ***S***tandard ***C***ode for ***I***nformation ***I***nterchange.

ASCII code is very similar to a secret code we may have invented as children. In this code, each letter (both lowercase and uppercase), number, and special character is given a sequential numeric representation. For example, the letter A is represented by the number 65; B by 66, C by 67, and so on. Similarly, the lowercase letters, a to z, are represented by the numbers 97 to 123, and digits 0 to 9 are represented by the numbers 48 to 57. All of the special characters have their numeric representations too. It takes one byte to represent a single character.

Appendix B contains a listing of ASCII codes.

When we issue a TYPE command at the DOS prompt (to type a file to the screen), the TYPE command must convert each ASCII code into the character it represents before typing that character on the screen. Similarly, the text we type to an editor is converted to the appropriate character codes prior to being written out to the disk.

How is this pertinent to the work we will do in AutoLISP? Some AutoLISP functions use the ASCII representation of characters rather than the characters themselves. Thus, we must look up the representations in the ASCII table, or call other AutoLISP functions that tell us what the ASCII representations are. In any event, it is definitely *not* worthwhile memorizing this chart!

All of our AutoLISP code and data are stored on disk in an ASCII representation, including numbers. The number 10, for example, is represented by the ASCII values for 1 and 0, 49 and 48. The computer is smart enough that when we ask to see the datum, it prints it out as a 10. However, before we can use it mathematically, it must be converted back to a number from its ASCII representation. Sometimes the computer does this for us, but other times we will have to do it ourselves. This is not hard, it's just something of which we must be aware.

Virtual Memory

Virtual memory is a technique for tricking the computer into thinking that it possesses more memory than it actually has. Recall that whenever we want to perform an operation on data, both the program and the data must be brought from the hard disk into main memory. A PC typically has several hundred mb of disk space, but much smaller amounts of memory.

As we do more and more work on the computer, we can eventually run out of memory. If that happens, AutoCAD freezes and we must reboot the machine, losing all of the changes that we made since the last time we saved our drawing.

Virtual memory alleviates this problem. When main memory fills up, AutoLISP automatically writes some data out to a special area on the hard disk. The data are written in units known as pages of data; thus the process is often called *paging*. The least recently used pages are written to disk, in the hope that we won't need that information soon.

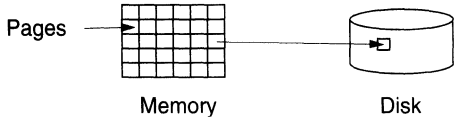


Figure 14-8

Once a certain number of pages have been written to disk, that amount of memory is freed up for us to work with. We can start creating more information and filling up memory once more. When memory becomes full again, more pages are written out to disk.

At some point we are probably going to need the information that is on disk. When we access data on a page that is no longer in memory, the least recently accessed data page in memory is swapped out (i.e., written to disk), and the page that we want is swapped back in. In this manner, the least recently used data are always on disk.

Figure 14-9

The benefit of virtual memory is that we are given the feeling of having unlimited memory to work with. Most of the memory is on disk (i.e., *virtual;* it's not really there), but this is transparent to us as users. Disk space is considerably cheaper than main memory.

The disadvantage is that storing and retrieving data on the hard disk is *much* slower than accessing data in main memory. We pay a stiff penalty in drawing speed every time the computer has to access the disk. A good paging algorithm can predict what data we're going to need, so that when a page is requested from disk it will bring in several at once. This minimizes the number of times that the disk must be accessed.

AutoLISP automatically pages our functions to virtual memory when it runs out of room in real memory. The manner in which this happens is discussed in the following section.

14.2.5 Advanced Topic: Memory Management

AutoLISP has several functions that enable us to manage our use of memory more efficiently. If you are new to AutoLISP, we suggest coming back to this section when you're more familiar with the language. However, if you are getting "insufficient node space" or "insufficient string space" errors, this section may help you overcome them.

AutoLISP objects are represented in 12-byte units of memory called *nodes*. To enable AutoLISP to manage these nodes efficiently, they are allocated from the

memory heap in groups called *segments*. Unless we alter its size, a segment consists of 514 nodes.

When a segment is allocated for our use, the nodes are added to a *free nodes list*. When we create a new symbol or function, or change a symbol's value, memory is drawn from this list.

AutoLISP might request some memory and learn that there are no nodes left on the free nodes list. When that happens, it performs a *garbage collection*. This process finds all nodes that are no longer being used and returns them to the free nodes list.

For example, assume that the value of the symbol `f` is a file descriptor. If we close the file and rebind `f` to `nil`, we can no longer access that file descriptor; the memory it occupies is considered garbage.

Here's another example. When we enter the form

```
Command: (* 3 4)
12
```

AutoLISP creates the list internally before sending it to `eval`. Once the list has been evaluated, it's no longer needed; it too is garbage.

So that our own programs might run the garbage collector, Autodesk has put the `gc` function at our disposal.

`gc`	Forces the garbage collector to run and returns `nil`.

In early versions of AutoCAD the garbage collector didn't run automatically. For this reason we frequently encounter `gc` calls at the start or end of older functions and commands. Currently we might want to run `gc` prior to loading an extremely large function or if we're running AutoCAD in a very small amount of memory. But nowadays this function is rarely called explicitly.

After garbage collection has been performed, there still may not be enough free nodes to fulfill the original request. When that occurs, AutoLISP procures another segment from the memory heap.

It might happen that the available segments are also used up. In that case *virtual function paging* is invoked. The least frequently used functions are paged out to virtual memory, and the space those functions occupied are returned to the free nodes list.

Prior to Release 12, virtual function paging wasn't performed automatically; the user had to invoke the vmon subr to initiate the process. Thus, many older functions contain a call to vmon at the start.

vmon	Turns on virtual function paging and returns `nil`.

Although it is still a defined subr, vmon is obsolete. It remains a part of AutoLISP so that older programs don't issue an error message.

Note: Although vmon is considered obsolete, it *does* have effects on the AutoLISP world. When it's invoked, a node called a *page table* is placed at the start of every function we subsequently write. The page table prints as a space and indicates that the function is swappable.

Consult the Memory Management chapter of the *AutoLISP Reference* for more information about the effects of vmon.

♦ ♦ ♦

Even with virtual function paging it is conceivable that memory could become full. When that happens AutoLISP issues an "insufficient node space" error. We must then exit and reenter AutoCAD.

This is a fairly severe measure, and if it recurs often, we must take precautions to avoid it. Our first step is to determine how many nodes are available.

mem	Prints current memory usage on the screen and returns `nil`.

```
Command: (mem)
Nodes:        6682
Free nodes:   306
Segments:     13
Allocate:     514
Collections:  14
nil
```

This is the meaning of the fields returned by the mem function:

Nodes	Number of allocated nodes, which is generally equal to the segment size multiplied by the number of segments

Free nodes	The number of nodes on the free nodes list
Segments	Number of segments allocated
Allocate	Number of nodes in a segment
Collections	Number of garbage collections performed during this AutoCAD session

We can control the number of nodes that are allocated with the `alloc` and `expand` functions.

`alloc` int	Sets the segment size to the specified number of nodes and returns the previous segment size.

```
Command: (alloc 100)
514
```

Our segment size was 514 nodes; now it's 100.

`expand` int	Requests the specified number of segments from the heap and returns that number.

```
Command: (expand 1)
1
```

One segment (100 nodes) is requested from the heap and added to the free node list. We can see these changes by running the `mem` function once again:

```
Command:    (mem)
Nodes:        6782
Free nodes:   397
Segments:     14
Allocate:     100
Collections:  14
nil
```

Our segment size is now 100, and we have 100 new nodes at our disposal. There are slightly less than 100 new free nodes because a few nodes were used by the `alloc` and `expand` functions.

We can place these functions in our `acad.lsp` file and regulate the memory allocation for our drawing session. However, we can't indiscriminantly assign huge amounts of memory to node space because the available amount is shared by strings.

Whenever a string is created, it uses memory from the heap. If string space becomes exhausted, AutoLISP issues an "insufficient string space" error. Thus, there can be a delicate balance between node space and string space, with a fatal error message arising from insufficient allocation of either. For this reason Autodesk suggests not altering the default memory allocations unless it's absolutely necessary.

Although node space refers to available memory, a node space error may also occur if the disk is overcrowded. If AutoCAD can't page to virtual memory, it will issue a fatal error regarding insufficient node space.

14.3 Writing and Running Programs

For those of us who have programmed for many years, writing programs is something we take for granted; we rarely stop and ask ourselves why or how we write them. But AutoCAD users who haven't programmed before don't always realize the advantages that programs provide or know how to create one. This section will explain these basics.

Before going further, let's differentiate several terms that are often used interchangeably:

Command	A directive typed to AutoCAD (as opposed to one supplied to AutoLISP). We can use AutoLISP to modify existing AutoCAD commands and to write our own.
Function	A piece of code we write in AutoLISP to perform a specific task. The AutoLISP language itself contains many built-in functions, such as addition and multiplication, which are called *subrs*.
Program	A collection of AutoLISP functions that work together to execute a more involved task. Because many of the routines that we write consist of only one function, those functions can be considered programs.
Routine	A somewhat "loose" term that is used to mean a *function, program,* or *command* that we have written.

File — A collection of information stored on a disk and given a name. The term *file* is sometimes confused with *program,* but they mean two different things. Programs can be stored in files on a disk. But there are also data files, drawing files, and many other kinds of files.

You must divest yourself of any notions you might have that a computer is a super brain. In fact, it has approximately the same intelligence as your toaster: It can't think at all. What it *can* do is execute, at unimaginably fast speeds, instructions that we give to it. This capability permits us to write programs that automate tasks we otherwise would have to do by hand. There is nothing magical about a computer; it is merely a tool to help us accomplish our work faster and more efficiently.

An excellent example of computer efficiency is payroll calculation. When your company computes your weekly pay, it has to take into account not only your weekly salary, hours worked, and tax deductions but also any vacation you took, how your sick time affected your Social Security payments, and so forth.

Processing this information for every employee of a large company is quite tedious. However, we can easily write a computer program to handle both the standard weekly steps and the small changes that occur for each employee from time to time. The program is a set of standard steps that we want to have continually repeated; we just need to plug in new numbers.

The same holds true for our drawings. If our drawing has 300 green holes and we want to make each one 1/4" bigger, then we might write a little AutoLISP routine to accomplish this task. If we only want to increase the size of one hole, it is probably easier to make the change in AutoCAD than it is to write an AutoLISP program. But AutoLISP can greatly curtail the amount of time it takes to make multiple changes to a drawing.

A main advantage to programs, then, is *repetition.* When we want to perform an action again and again, it becomes advantageous to write a program to do so.

Suppose we want to create a hundred identical flanges. Even though this task is repetitive, you might argue that it's easier to create one flange, then block and move it, or array it around the drawing. Repetition, therefore, is not the sole criterion for writing a computer program.

Let's say, however, that each of these flanges is going to be subtly different. Perhaps we'd like the operator to supply a different web for each of them. Wouldn't it be nice to have an AutoCAD command that would draw flanges and

ask the user for the information that it needed? Well, we can easily write such a command in AutoLISP and put that command on our tablet or in a menu.

Thus, the second major advantage to writing a computer program is *diversity.* Programs have the ability to perform the same or similar steps on different data of the same kind. By "same kind" I mean that this program will work on any flange but will not work with a piston in a car. We can write an AutoLISP function or AutoCAD command that performs a common task and prompts the user for any information that it needs.

Many drafters begin a new drawing by creating a title block. We can write an AutoLISP function that prompts the user for the necessary information, such as the Job Title and Revision, then creates the appropriate title block. Because the operator only needs to type in the required data and never has to do any drawing, he can create a very elaborate title block in practically no time.

Every time we create a title block for a new drawing the steps are exactly the same; only the data change. This brings us to the definition of a computer program:

> A computer program is a series of instructions to the computer. These instructions are repeated again and again, but on different data.

The important point is that the steps usually remain the same and the data typically change. Keep in mind the title block example: Every drawing will have a different Job Title and Date, and some won't even have a Revision. But we execute the same steps to get the title block into the drawing.

One further advantage that AutoLISP has over AutoCAD itself is the ability to modify the entity database. Every time we create an entity, such as a line, an arc, an attribute of a block, or even a block itself, AutoCAD puts information about that entity in a database. We cannot access that information in AutoCAD alone. But AutoLISP can access the database to determine starting and ending points of lines, polyline widths, text height, and so forth.

In many cases we can even change the data in our drawings. In this book we'll write functions to change the color and layer of selected entities, alter attribute text, scale multiple blocks, and many, many more.

Keep in mind the benefits computer programs provide as you learn how to write them. Then you'll better understand why we write code as we do, and use the language more efficiently.

♦ ♦ ♦

As we shall see in a more practical sense when we begin to write our own functions, we must differentiate between the time we *write* a function or program and the times that we *call* that function or program.

When we create a program to calculate a company payroll, this program is written only *once*. Naturally, we'll have to modify the program occasionally, such as when new tax laws are passed. But in general, once a program is working correctly it remains a process that we can depend on to behave the same every time it is used.

After it has been written, this program may be called many times. We might run a payroll program weekly for every individual in the company. Each time we run this program the data will probably change, but the program's steps remain the same.

Another example is an AutoCAD command. Every time we execute a CIRCLE command we can supply a new center point and radius, but the action the command takes is the same: It always draws a circle, never an arc or an ellipse.

In any computer language a program is written only once, but it's called innumerable times thereafter.

14.3.1 The Edit-Compile-Debug Cycle

When we write programs in a standard computer language, we go through a process known as the *edit-compile-debug* cycle. AutoLISP is an unusual language in that we are able to bypass some of the steps in this process. It is helpful to have a cursory understanding of the standard way programs are written and debugged, as compared with AutoLISP's simpler approach.

Program creation usually involves the following tasks:

1. Determine what we want the program to do. What data will it use? What actions will it perform? What result will it return?

2. Write down the steps, in the computer language, on a piece of paper. If the program is particularly long or involved, we might flowchart it first or write an English language description. On the other hand, if the program is very simple and we have a good editor, flowcharting is probably unnecessary and we might even write our program directly into the computer.

3. Type these steps into the computer and save them as an ASCII file on disk. Although we can enter AutoLISP programs directly at the Command:

prompt, we typically use an editor to enter the program and save it away. Otherwise, the program is lost when we exit our AutoCAD drawing.

4. Compile the program. This task is not necessary in AutoLISP.

5. Load the file into memory and run the program.

6. Once run, we almost always find errors. Correct these errors (*debug* the program) and repeat steps 3 to 6 until the program works.

Let's examine these steps.

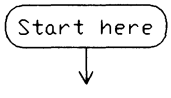
Flowcharting

It is frequently beneficial to describe our code before we try to write it. In many cases, an English-language definition of our task will suffice. Sometimes, however, the task is too complex to set forth succintly. Flowcharting is a way of describing a task through pictures rather than longhand.

It is not our aim to make you a flowcharter. However, we utilize flowcharts in this book to describe how certain AutoLISP functions work, so you should understand the meaning of the different pictures that we use.

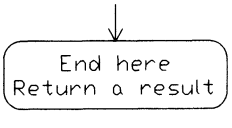
Arrow

Indicates the direction of code flow. Always move through the flowchart in the direction indicated by an arrow.

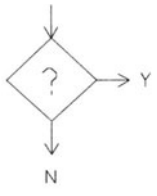


Terminator box

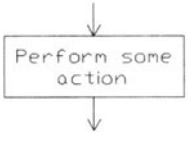
Indicates an entry or exit point in the function. If an arrow is coming out of a terminator box, it indicates an entry point into the function. There is typically only one entry point, and data are passed into the function at that time.



If one or more arrows are coming into a terminator box, then it indicates an exit point out of the function. There may be several exit points, and a result is often returned at this time.



Decision box — Used to ask a yes-no or true-false question. This box may have several entry points but always has exactly two exit arrows. If the answer to the question is "yes" or "true," then the path marked "Y" is taken; if the answer is "no" or "false," then the path marked "N" is taken.



Process box — Specifies some work to be performed.

This flowchart describes the steps needed to print numbers from 1 to N, where N is a number supplied to the program.

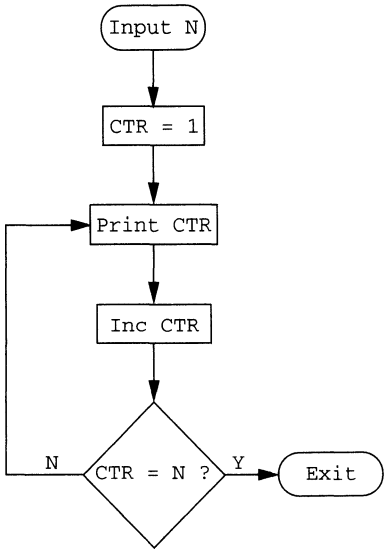


Figure 14-10

After receiving the number, N, the program initializes a counter, CTR, to 1. It then runs a loop (a series of repetitive steps) to print the numbers. Each time through the loop it prints and increments (increases by 1) CTR, which contains the value we want shown on the screen. It then checks to see whether CTR is equal to N. If it is, the program is done, so it exits; if not, the loop is executed again.

Editing Programs

By now you are probably familiar with one of the hundreds of text editors available. If not, it is very important that you learn how to use one. Some editors are freeware, some cost a minimal amount, and others are quite expensive because, like WORD, they are word processors that do much more than simply process ASCII text. Most word processors are more sophisticated than is needed for writing AutoLISP programs. Everyone who uses DOS automatically has the DOSEDIT or EDLIN editors. They are rather cumbersome to use, but at least they are editors that you own.

Someone at work probably uses an editor that she really likes and will be happy to show you how it works. Alternatively, there are two editors on the supplied floppy disk that you are welcome to use. The DOS-based AHED editor is explained later in this chapter. The Windows-based PFE editor is discussed in the `readme.txt` file.

Recall that all work is performed on data that are in memory. Memory, however, is volatile, and we lose what is in it when we turn off the computer (or, in the case of AutoCAD, when we exit our drawing). What we need to do is write our program in main memory, then save that program onto the disk, because the disk is a permanent storage device. In that way our work is saved even after we turn the computer off.

This is what an editor does for us. It allows us to write a program, give the program a name, then save that program on disk. It also lets us bring an already existing program into memory (via its name) so that we may modify it. And a good editor will contain wonderful tools that facilitate the process of writing a program.

Compiling/Interpreting, Loading, and Debugging

Most programming languages require that we perform a process known as *compilation* of the code; AutoLISP does not. Let's examine how this process is performed and how it affects us as programmers.

The set of instructions that we type into the computer may look like a program to us, but, unfortunately, it is in a format that the computer cannot understand. The code is stored on the disk as ASCII text, but the computer understands 0s and 1s at an even lower level than that. Unlike AutoLISP, most conventional computer languages require us to translate the ASCII text into the 0s and 1s that the computer understands.

The program that performs this translation is called a *compiler.* When we supply to the compiler the name of our program (called the *source file*), it gets the instructions that we typed in and converts them to 0s and 1s. It then writes back to disk a version of our original file that has been translated into this binary form. This is called the *object file.* We cannot read or understand an object file, but the computer can. We must load this file into memory from the disk, and only then can we finally run our program. These steps are shown in Figure 14-11.

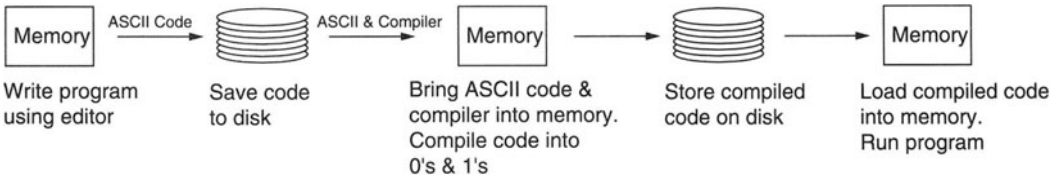


Figure 14-11

Pretty complicated, eh? Fortunately, in AutoLISP we are able to bypass the compilation step. After we save our program onto disk with the editor, we simply load that program into memory. The AutoLISP loading program performs a process called *interpretation,* which changes our ASCII text into data that the computer understands *without requiring us to go through the compilation stage.* This is partly because AutoLISP does not work directly with 0s and 1s; it works with a higher-level data representation known as AutoLISP objects. The loading program merely has to translate the text we type into these AutoLISP objects, a much simpler task than compiling down to 0s and 1s. These steps are shown in Figure 14-12.

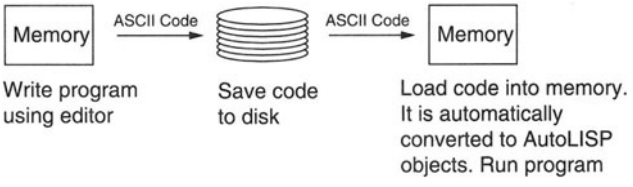


Figure 14-12

A good compiler will optimize the code it is compiling, enabling the compiled code to run faster than interpreted code. This is the central reason that Autodesk added hooks to the C language (whose programs *are* compiled). Common Lisp, a more general version of Lisp than AutoLISP, has the ability to compile code. It is hoped that before long a good AutoLISP compiler will be available as well.

Here is one outstanding feature of AutoLISP that will become abundantly clear as we start to program: Because AutoLISP is an interpretive language (i.e., does not require the compilation step), we are not required to enter our code using an editor. Instead, we can enter AutoLISP code directly to the Command: prompt in AutoCAD. This means that any AutoLISP code we type at the Command: prompt is executed immediately, with a result given to us on the spot.

Type code directly to Command: prompt.
Code is run immediately

Figure 14-13

Interpretation has a couple of distinct advantages over compilation:

- Compilation is a very time-consuming step. Programmers often write large programs rather than small routines so that they don't need to compile as frequently. Unfortunately, when large programs have errors, it generally takes a long time to locate those errors and debug the code.

  The AutoLISP style is to write short programs, debug them, then add to those corrected routines. It is usually much faster to fix a 10-line program than it is to find an error in 250 lines of code.

- If we need to understand what someone else's program is doing, we can type in the program one line at a time and watch it run. It is considerably easier to comprehend a program by analyzing its constituent parts than by having to run the entire program at once.

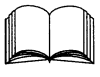
  This capability is one of the primary reasons that Autodesk chose Lisp as the language for AutoCAD. The advantages will become more obvious after we actually learn to program in the language, so you may wish to reread this section in the future.

Entering code at the Command: prompt is very convenient and useful for testing purposes, but everything we type there is lost when we exit our drawing. Ultimately, we must enter our programs using an editor and save them away on disk. Otherwise, each time we start a new drawing we will have to retype the code. However, we can still load and run our code without needing to compile it.

The last step, *debugging,* is the process of finding and correcting our errors (*bugs*). When we find even one error we must go back to the program and,

using our editor, make the necessary corrections. We then load the corrected file into memory, and rerun the program. We must repeat these steps until all of the errors have been fixed.

One of the central aims of this book is to provide you with an understanding of how the AutoLISP language is structured, and to present effective ways of writing AutoLISP code. Programmers who understand these fundamentals make fewer errors in the first place and can fix their bugs in a minimal amount of time.



Chapter 12 contains a study of common AutoLISP error messages and various debugging techniques.

14.3.2 Calling Functions

When we want to accomplish a task in AutoLISP, we call a function to perform the necessary actions. Sometimes a function gets to a point where it is blocked, because it needs some information it currently lacks.

Typically, another function can get the information that our function needs. If we have set up our function correctly, it will know to call that other function and ask it to compute and return the needed information.

All computer languages are capable of asking one function to call another. In traditional languages, like FORTRAN or C, the programmer creates a main routine that does most of the work. If the main routine needs additional information, it can call a subroutine to find the appropriate data.

AutoLISP doesn't have the notion of "main routines" and "subroutines"; we call all of its routines *functions*. All functions are on equal footing, and any function can call any other function; there is no hierarchy. The actions taken are pretty much the same, but the structure is more flexible in AutoLISP than in traditional languages.

Suppose we run a function called `f1`. We'll indicate the path of the function's execution by a squiggly line.

Figure 14-14

At some point `f1` realizes that it can't continue without getting more information. It knows that the function `f2` can get that information, so it calls `f2` and probably passes it some data to work with. Function `f1` stops running right at this point, and `f2` begins to run:

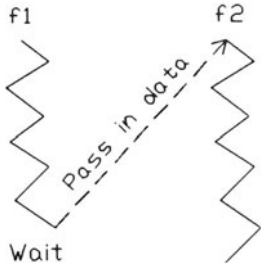


Figure 14-15

When `f2` completes execution, it hands its result back to `f1` and `f1` runs to completion:

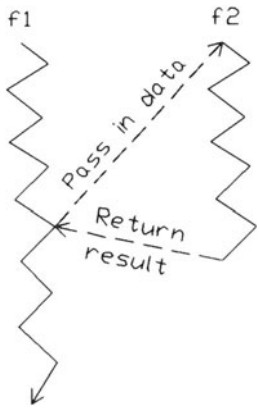


Figure 14-16

Now, suppose `f2` gets blocked. Then it might call another function, `f3`. In this case, both `f1` and `f2` wait while `f3` runs:

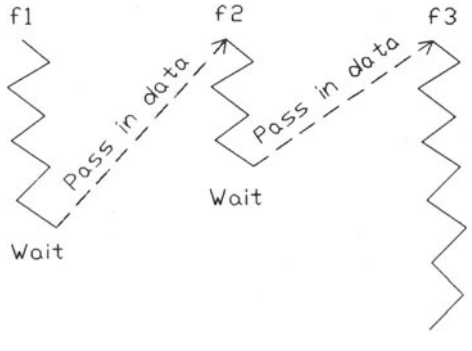


Figure 14-17

When `f3` completes its task, it returns its result back to `f2`. `f2` runs to completion and returns a result back to `f1`, and now `f1` can finish its task and return a result back to us:

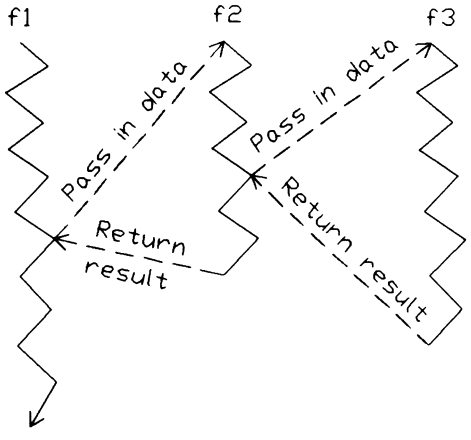


Figure 14-18

Here are the salient features of function calling:

- When one function calls another, it typically passes data for the called function to process. The calling function then pauses and waits for the called function to complete execution.
- The called function begins running at the start of its code. Functions never commence execution from the middle.
- When the called function completes its task, it returns control (and, usually, a result) back to the calling function. Every function always returns a result back to whoever called it. If `f1` calls `f2` and `f2` then calls `f3`, `f3` returns its result back to `f2`; it never returns its result to `f1`. Function calls are nested one inside the other, so `f3` knows nothing about `f1`.
- When control is returned to the calling function, it continues processing from the point where it issued the call to the called function. It does not start running from the beginning.

Let's look at a real-life analogy to the function-calling process. Suppose you wrote a report for presentation to your group. Before your meeting you want to get copies of your overheads to hand out. So you approach your group's secretary and ask him to make 20 copies.

At this point you are blocked: You cannot give your presentation until you get those copies back. You have made a call to your "secretary function" and must wait until you get your results (i.e., the copies of your overheads).

As your secretary is making the copies, the photocopier breaks (I know this never happens at your company, but try to imagine the situation!). The secretary must call the repair person and wait until the copier has been repaired. This is analogous to `f2` calling `f3`; the "repair person" function now gets to run while you and your secretary wait.

When the repair person has fixed the machine, she informs the secretary. He regains control of the process and picks up where he left off making copies. Note that he does not start from the beginning.

When the copies are made he gives them to you, and you can finally go to that meeting (you're a bit late because the copier broke, but you shouldn't have waited until the last minute!).

Looping

As we saw earlier, the main advantage of a computer program is its ability to rapidly repeat a set of steps. Suppose we write a function to size a door and put it into a doorframe. If a building has 65 doors, we don't want to have to call this function 65 times. Instead, let's have the function itself determine how many doors are needed and fit them into the proper doorframes for us. Depending on the situation, we can write a program that repeats its steps iteratively or recursively.

A *loop* is a section of computer code that iterates (continually repeats) a prescribed set of functions. The ability to loop is common to all computer languages.

AutoLISP provides three different kinds of loops. A `repeat` loop repeats its task a specified number of times; a `while` loop repeats its steps until some end condition is met; a `foreach` loop repeats once for each element in a list and allows us to access and modify that element.

All loops follow a similar pattern. At the start they initialize a counter or a test that will move closer to some goal during each pass through the loop. At the conclusion of every pass the counter is modified to bring it closer to that goal, or the test is performed to see if the termination condition is met. Each loop has a set of steps that specify the work we want performed. These steps are repeated until the stated goal is met.

Going back to our photocopying example, you might tell your secretary to make 20 copies for the meeting. We can create a flowchart for a generalized secretary function that would make N copies of the report:

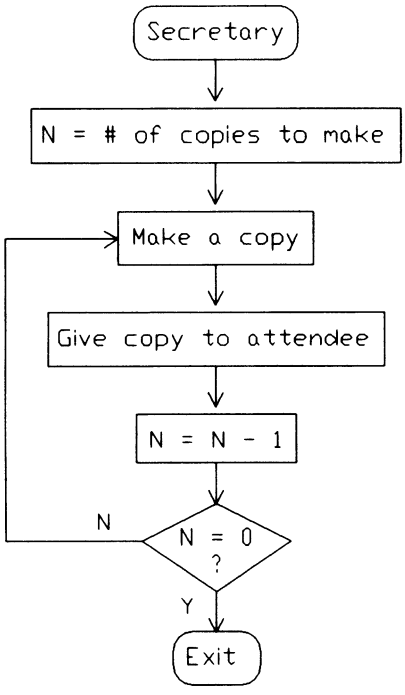


Figure 14-19

Each time through the loop, the secretary makes and distributes one copy, and the loop counter, N, is decremented (i.e., decreased by one). When N reaches 0, the function is done and it exits.

When we create a loop, we must incorporate some way for that loop to terminate. Otherwise we have what's known as an *infinite loop:* The program will run but the screen won't change, and it will look as if the computer has frozen. For example, we could tell a loop to repeat until it locates a line with an endpoint beyond the value of LIMMAX. But if no line fulfills that requirement, the loop won't end. When this happens we have to press <esc> to abort the function.

AutoLISP's iteration functions minimize the overhead of managing a loop; once we've set one up we don't have to decrement or increment counters. However, if you're just beginning to program, you might find it easier to write the set of steps *without* the loop first, get those to run, then add the looping mechanism afterward. Until the steps run correctly one time, there's no advantage to having them repeat.

Recursion

Another approach to repeating a set of steps is to write a recursive function. A *recursive function* is one that calls itself. Some tasks that require repetition are easier to solve by writing recursive functions than by writing a loop to repeat the code. For example, the DRAW program that is introduced in Section 2.7 was more easily written in a recursive fashion because it has to be able to draw lists that are nested arbitrarily deep.

More significantly, the most important function in the language, `eval`, is a recursive function. Chapter 4 is devoted to an explanation of this essential function, so it's imperative that you understand the concept of recursion.

An initial reaction upon encountering this concept is to exclaim, "If I write a function that calls itself, the function won't ever end!" Well, a recursive function that never ends is known in programming parlance as a "badly written recursive function!" It's rather like an infinite loop that keeps looping and never stops.

Somewhere in its body of code, a well-written recursive function will always ask the question, "Am I done?" If the answer is "No," it takes a path that causes it to call itself another time and makes a modification that will bring the test closer to succeeding the next time it is run. If the answer is "Yes," it takes a path that leads to the function's conclusion.

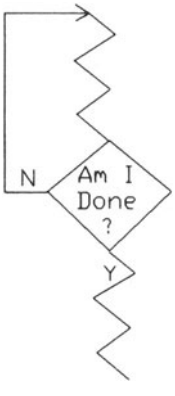


Figure 14-20

There might be several levels of recursive calls. When the lowest level is done, it returns a result back up to the next level. That level finishes processing and sends a result back up another level. The recursion unwinds in this manner until the first level finishes running and returns the final result:

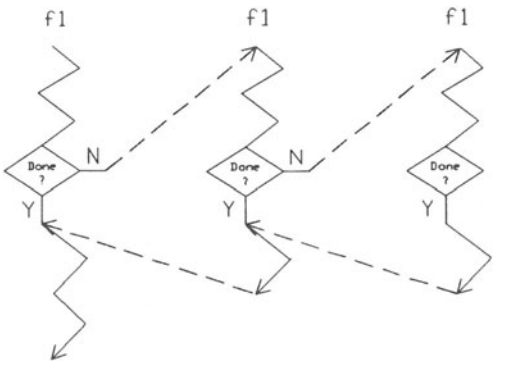


Figure 14-21

Figure 14-21 shows the program flow during recursive function calls. The only real difference between this picture and standard function calling shown in Figure 14-18 is that the same function is repeatedly called. The actions taken by AutoLISP are pretty much the same.

The majority of AutoLISP programmers don't use recursion very often. Most of the time when repetition is needed we write a loop. But sometimes, as with `eval`, recursion is the appropriate tool.

The techniques for writing recursive programs are shown in Section 6.4.

14.3.3 The AHED Editor

On the floppy enclosed with this book is an editor named AHED. AHED was written by Michael Covington at the University of Georgia. It is a small, easy-to-use, DOS-based editor with on-line help facilities. Furthermore, it has a couple of features, such as highlighting matching parentheses, that make it particularly useful for AutoLISP. Even if you are currently using an editor, you may find that you prefer AHED.

See the `readme.txt` file on the enclosed floppy for information regarding the supplied Windows-based editor, PFE.

To use the editor, copy it into your current directory. At the DOS prompt (shown as well) type

```
>ahed file.ext
```

where `file.ext` is the filename and extension of a file you wish to create or modify. If the file does not exist, AHED indicates that it is creating a NEW file.

All you need to remember about AHED is to look in the upper right-hand corner of the screen. There it says, "Press F1 for Help". When we press `F1`, the menu shown in Figure 14-22 appears. When we enter any of these letters, another menu pops up on the screen and displays commands related to the selected topic. For example, if we type "C" it displays a menu of all the cursor movement commands available. Many of these cursor commands are intuitive and permit movement via the arrow keys.

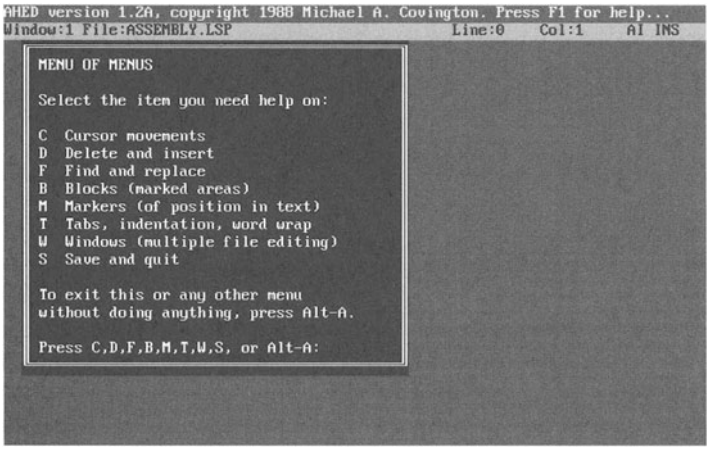


Figure 14-22

AHED has several characteristics that make it a nice editor for use with AutoLISP:

- When the cursor is on or to the immediate right of a right parenthesis, AHED highlights the matching left parenthesis. AutoLISP uses so many parentheses that you shouldn't consider an editor that does not in some way indicate matching parentheses.

- When we press `<enter>` to go to the next line, AHED positions the cursor underneath the first character on the previous line, rather than at the left margin. AutoLISP has very strict conventions regarding indentation and, although AHED doesn't indent for AutoLISP, it comes close enough for this to be a useful feature.

- AHED is quite small. The entire program is a mere 63K and the associated `.doc` file is another 10K.

- AHED is free! This editor is freeware that may not be sold. However, the author requests that you *not* contact him about the editor unless you discover a bug in it.

AHED has a couple of handy features that are present in many, but not all, editors. Note that AHED documentation uses the carat (^) to indicate the `Ctrl` key and the at-sign (@) to indicate the `Alt` key. This explanation will utilize that standard. ← and → indicate the left and right arrow keys.

- We can create a *region* or *block,* then copy or move that block somewhere else in the file. To create a block, place the cursor where you want the block to start and enter `^K^B`. Next, place the cursor where you wish the block to end and enter `^K^K`; the block will be highlighted. Thereafter, move the cursor and enter `^K^V` or `^K^C` to move or copy the block to the current cursor position. `^K^Y` deletes the block and `^K^H` unblocks the text.

- We can split the screen into two or more windows by typing `^O^O`. Thereafter we can put a different file in each window and move text between files. Or we can put the same large file in both windows and look at different parts concurrently. However, AHED will not support very large menu files.

The arrow keys work in standard fashion, as do `<Pg Up>`, `<Pg Dn>`, `<Backspace>`, and `<Del>`. Following are the more commonly used AHED commands:

F1	Show menu of menus
F2	Save file and continue
F3	Save file and exit
F4	Quit without saving the file
^K^W	Save file with another name
^K^R	Read in a file
←	Move left 1 character
→	Move right 1 character
^←	Move left 1 word
^→	Move right 1 word
<Backspace>	Delete left 1 character
<Del>	Delete right 1 character
^T	Delete right 1 word
^Y	Delete current line
<esc>	Undelete a deleted line
^N	Insert a blank line

Table 14-5

^Q^F	Find a string
^L	Find next occurrence of that string
^Q^A	Find and replace
^K^B	Mark beginning of block
^K^K	Mark end of block
^K^C	Copy block to cursor
^K^V	Move block to cursor
^K^Y	Delete block
^K^H	Hide block (i.e., unblock)
^O^O	Create window
^O^Y	Remove window
^O^X	Jump to next window
@A	Abort the current command (like <esc> in AutoCAD)

Table 14-5 (*continued*)

14.3.4 The `acad.pgp` File

The `acad.pgp` file contains information that allows AutoCAD to run external DOS programs. When we issue a command that AutoCAD doesn't recognize, it searches the `acad.pgp` file to see if it's there. In short, it enables us to run DOS programs at the Command: prompt without shelling out.

The following is a partial listing of the standard `acad.pgp` file:

```
CATALOG,DIR /W,     0,File specification: ,0
DEL,DEL,            0,File to delete: ,4
DIR,DIR,            0,File specification: ,0
EDIT,EDIT,          0,File to edit: ,0
SH,,                0,*OS Command: ,4
SHELL,,             0,*OS Command: ,4
TYPE,TYPE,          0,File to list: ,0
```

You'll notice that EDIT is among the standard AutoCAD external programs. If you have an editor that you prefer, you can add it by editing this ASCII text file. For example, we can enable the use of the AHED editor in AutoCAD by adding the following line to the `acad.pgp` file:

```
AHED,AHED,          0,File to edit: ,4
```

The fields in the `acad.pgp` file are separated by a comma. Let's look at the information needed for an `acad.pgp` entry:

Command name	What the user will enter at the Command: prompt.
File command	The name of the program we want run. For example, when the user types EDIT, the DOS EDIT program is run.
Memory	Required, but always 0. Prior to Release 12 it specified the number of bytes that the program needed.
Prompt	A string that is printed on the screen when this command is issued. If the first character is an asterisk, then the user's response can contain spaces.
Return code	An action to be performed upon conclusion of the command. Typical settings: 0—Return to the text screen 4—Return to the screen being used when command was issued

The File command, Prompt, and Return code fields are optional. If omitted, the comma delimiters are still required.

The following two commands are also handy to have in our `acad.pgp` files:

```
DOSCOPY,COPY,0,*File to copy: ,4

FORMAT,FORMAT A:,0,,4
```

The former allows us to run the DOS COPY command in AutoCAD by entering the command DOSCOPY. When prompted, we supply both filenames exactly as we would in DOS. We can't call this command *COPY* because there already is an AutoCAD command with that name.

The latter automatically formats a diskette on drive a:, thereby preventing us from entering, say, `Format c:` and accidentally destroying what's on that drive. We may never intentionally call this command, it's merely there for protection. If you're a system administrator, you might want to add this line to your users' `acad.pgp` files.

The acad.pgp file is loaded along with AutoCAD. Thus, if we modify it from within AutoCAD, the changes won't take effect until the next time we run AutoCAD. When using versions prior to Release 12, we must exit AutoCAD entirely and get back in.

Release 12 introduced a command called REINIT that enables us to reinitialize the acad.pgp file without exiting AutoCAD. Simply enter REINIT at the Command: prompt and select the *PGP File* option in the dialog box that appears on the screen.

Windows Note: Some DOS programs, such as TYPE and DIR, cannot be run from Windows. You must run SHELL, press <enter> to shell out completely to DOS, then run the standard DOS command. Thereafter, type EXIT to return to AutoCAD. The *examples.lsp* file on the enclosed floppy contains the c:printit command, which emulates the TYPE command in Windows. See Section 8.3 to learn how to use this command.

In addition to the specification of DOS programs, the acad.pgp file also contains command aliases. *Aliases* are abbreviations for AutoCAD commands, such as the following:

```
A,    *ARC
C,    *CIRCLE
CP,   *COPY
```

An alias entry consists of the command alias followed by a comma, an asterisk, and the command name itself. Once the alias has been created, we can refer to the command by its alias rather than by its full name. Thus, instead of typing *CIRCLE,* we can simply enter *C.*

These aliases, plus others, are already defined in the acad.pgp file. We can alter these, add our own, and also create aliases for AutoCAD commands that we write.

14.4 Labs

1. Which of the following statements are true, and which are false?

   a) A program must be in memory before it can be run.

   b) AutoLISP programs are stored as binary (base 2) files.

   c) When you enter a cafeteria and take a tray from the top of a collection of trays, it is correct to say, "Here's a good example of a queue."

   d) An advantage of computer programs is that they repeat the same actions on the same data.

   e) We can use AutoLISP to modify the entity database, but in AutoCAD we're denied direct access to this database.

   f) It is always necessary to enter AutoLISP code via some editor.

   g) If we use an editor to change a program, then that program must be reloaded before AutoLISP knows about those changes.

   h) When we exit our drawing the information in memory is lost, but information on the hard disk is retained.

   i) AutoLISP can actually modify the built-in AutoCAD commands.

   j) On any computer there are 8 bits in a byte.

   k) There are 1000 bytes in a kilobyte.

   l) Although flowchart arrows point a certain way, it is occasionally necessary for the flow to move in the reverse direction.

   m) A function always returns results to whoever called it.

2. Fill in the blanks:

   a) The "brain" of the computer is called the _____. Computers that have more than one of these can perform _____.

   b) Some computers give the appearance of running several programs at one time. This is called _____.

   c) Information on a hard disk can be accessed in any order that we choose. Thus the disk is called a _____ _____ device. Magnetic tape, on the other hand, accesses its information in the other in which it was written. It's called a _____ _____ device.

   d) Examples of output devices are _____ , _____. Examples of input devices are _____ , _____. Examples of auxiliary storage devices are _____ , _____.

   e) Assume we store, in order, the numbers 1, 2, and 3 on a stack. The first number popped will be _____. If these same numbers are placed on a queue, then the first number retrieved will be _____.

   f) Two reasons we write functions with an editor rather than simply entering the code at the Command: prompt are _____ and _____.

   g) The process of finding and correcting errors in our code is called _____.

   h) A program that changes programs we write into a file containing 0s and 1s is called a _____.

   i) When memory fills up, the least recently accessed information is sent to _____ _____.

   j) A _____ function calls itself.

Appendix

A

Lab Answers

This appendix contains the solutions to all of the lab exercises in this book. Here are some important points:

- You will gain more from a lab exercise if you try your best to solve it before consulting the solution.

- All of the programs shown here are in the `answers.lsp` file on the enclosed floppy. If you're not sure what you're supposed to write for a particular exercise, run the program and see what it does. If a function appears helpful in your work, use it and be more productive!

- Once you have written a function it is well worth your while to examine the solution in this appendix. Very often there are several ways to arrive at the same result, and we frequently supply two or even three different solutions to the same exercise. Furthermore, there's occasionally an explanation along with a solution that will enrich your understanding of AutoLISP.

Chapter 1. The AutoLISP World

1. Which of the following are *not* valid numbers in AutoLISP? Why?

   b. `2S` This is a symbol.

   f. `"654"` This is a string.

   g. .2 Real numbers entered to AutoLISP that are less than 1 must have a leading 0.

   h. 3/4 Fractions are not allowed. This would be taken as a symbol.

   i. 3,4 This is a point in AutoCAD.

2. Which of the following are *not* valid symbol names? Why?

   a. 56 This is a number.

   b. `'xyz` Symbol names can't contain a quote.

   d. `(x y)` Symbol names can't contain parentheses. This is a list.

   h. c.34890a Symbol names can't contain a period.

   j. `long symbol name` These are three separate symbols. We must separate multiword symbol names with hyphens: `long-symbol-name`

3. The values of `x`, `y`, `z`, and `w` are 5, `nil`, `w`, and 3, respectively.

4. a) The value of `p` is the cons whose car is `"Roof"` and whose cdr is 7.

   b) The cdr is 7. It's a number.

   c) The car is `"Roof"`. It's a string.

5. a) The value of the symbol `q` is the cons whose car is `x`. Notice that since conses don't have names, it is very difficult to pinpoint them. We often have to describe them in terms of their cars or cdrs.

   b) The car is `x`. It's a symbol.

c) The cdr is the cons whose car is `y` and whose cdr is `z`.

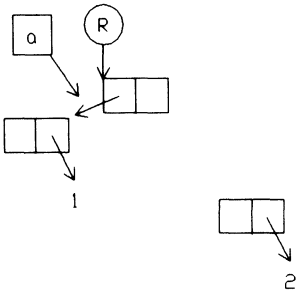
d) Because no value is shown for `x`, its value is `nil`. The values of `y` and `z` are 6.6 and `y`, respectively. Note particularly that `z`'s value is `y`, not 6.6; we *don't trace through arrows!*

e) The value of `q` is a cons, the cdr of that cons is another cons, and the car of *that* cons is `y`.

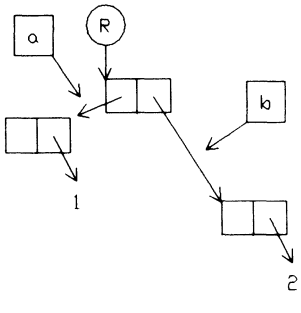
f) `z` is the cdr of the cdr of the value of `q`.

6. For this exercise, the lowercase letters in boxes indicate which line was drawn by each step.

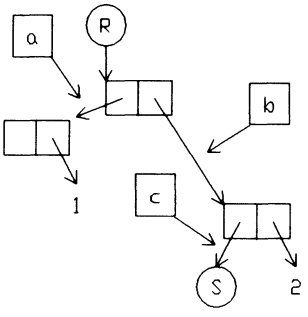
a) The value of the symbol `r` is the first cons. The cons on the lower left has a cdr of 1. Step *a* makes the car of the first cons be the lower left cons.



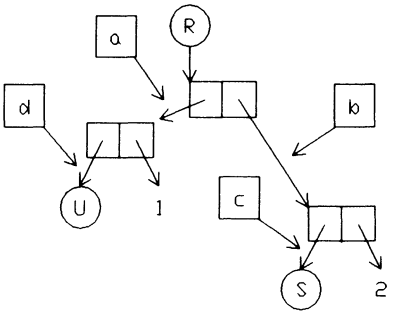
b) The cons on the lower right has a cdr of 2. Step *b* makes the cdr of the first cons be this cons.



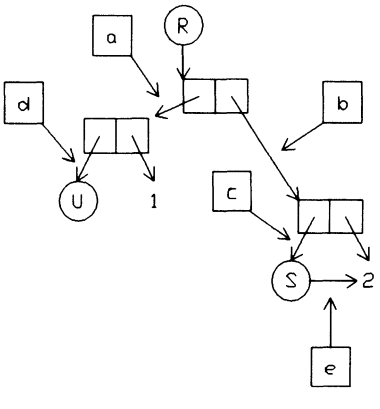
c) The cons on the lower right is the cdr of the first cons. Step *c* makes the car of this cons be the symbol `s`.



d) Step *d* makes the car of the lower left cons be the symbol `u`.



e) Step *c* determined that `s` is the car of the cdr of the first cons. The cdr of the cdr of the first cons is 2. Step *e* makes the value of `s` be 2.



7. a) The value of `r` is a list. Its elements are `a`, `b`, `c`, `d`, and `e`.

   b) The value of `s` is a list. Its elements are `c`, `d`, and `e`.

8. a) The value of `m` is a list. Its elements are `a`, `b`, the cons whose car is `1`, and `c`.

   If you said that 1 is a member of this list, you traced through arrows. The *cons* whose car is 1 is the actual member because it is the car of the backbone cons.

   b) The value of `n` is a list. Its elements are `1` and `2`.

9. a) The value of `parts` is a list. Its elements are `x1`, the cons whose car is `"y1"`, and `x3`. The symbol `x2` is not a member of any list.

   b) The value of `x2` is not a list. It's a cons whose car is `"y1"` and whose cdr is `"y2"`.

10. a) This cons is indeed a list, and its only element is `nil`. Why would we ever have `nil` as a member of a list?

    Although we would never need a list whose sole element is `nil`, `nil` is often used as a placeholder in a larger list. Suppose we have lists representing each employee in our company, and the third element of each employee list is the name of the employee's spouse. If the employee is unmarried, we might make the third element of that list be `nil` to indicate that there is no spouse.

    Similarly, if a list element represents information about the windows in a room, what do we put for an interior bathroom? Well, we might use 0 to indicate that there are no windows, but it is often more useful to have `nil` in that spot.

11. a) The cars of the conses labeled A, B, and C are `lynn`, another cons, and `jillian`.

    b) The cdrs of these conses are `nil`, another cons, and 8.9.

    c) The values of `jillian` and `lynn` are 8.7, and `nil`. `"Emily"` is a string and therefore does not have a value.

    d) There are two elements in this list. They are both lists.

e) The symbol `lynn` is the car of the car of the value of `kevin`. We can access `lynn` as follows:

```
Command: (car (car kevin))                    ;or (caar kevin)
LYNN
```

f) `jillian` is the car of the car of the car of the cdr of the value of `kevin`.

```
Command: (car (car (car (cdr kevin))))        ;or (caaadr kevin)
JILLIAN
```

8.9 is the cdr of the car of the car of the cdr of the value of `kevin`.

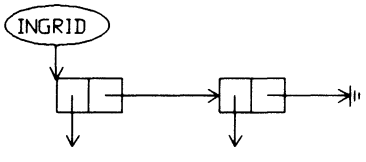
```
Command: (cdr (car (car (cdr kevin))))        ;or (cdaadr kevin)
8.9
```

`"Emily"` is the car of the cdr of the car of the cdr of the value of `kevin`.

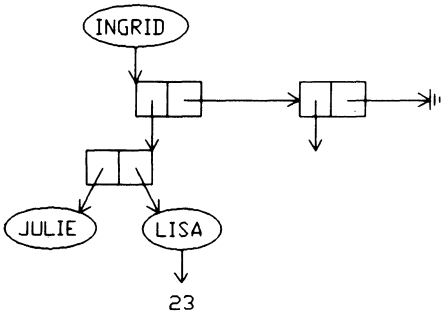
```
Command: (car (cdr (car (cdr kevin))))        ;or (cadadr kevin)
"Emily"
```

12. The conses drawn in this exercise should appear as follows:

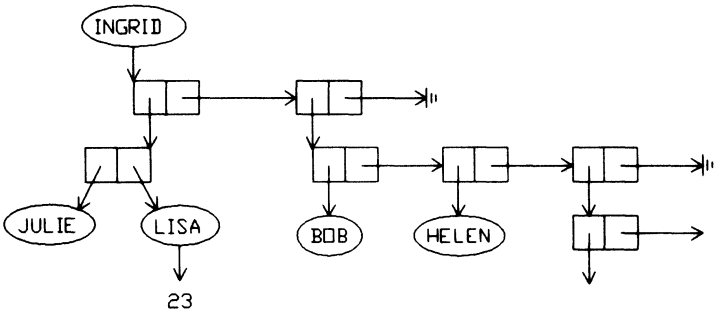
a) A list of length two.



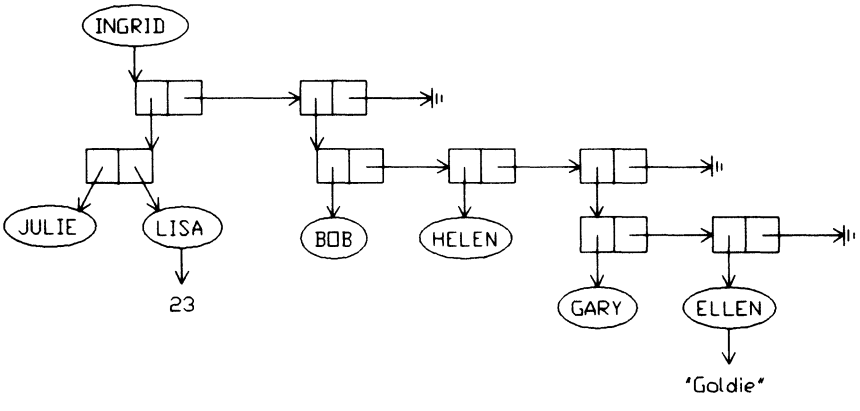
b) The first element is a cons.



c) The second element is a list.



d) This is a list of two elements.



Chapter 2. Printed Representation of AutoLISP Objects

1. a) `((larry . 33) (magic . 32) (michael 23 . 45))`

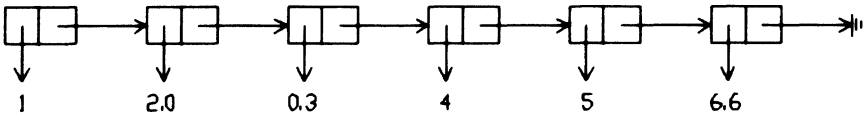
   b) `(1 (2 3 4) (5 6))`

   c) `((a) (b) (c . d))`

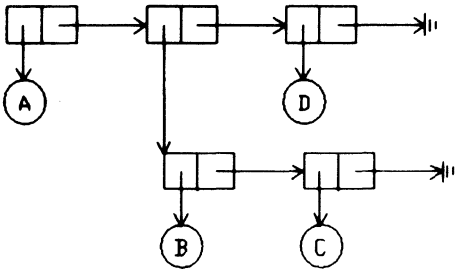
   d) `("First" ("Second" ("Third" "Fourth") "Fifth") "Sixth" "Seventh")`

   e) `(((z 4.4) y 6.2) x 3.3)`

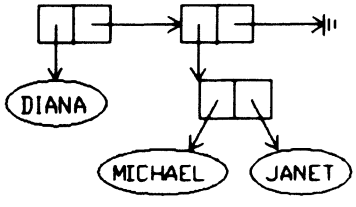
2. a)



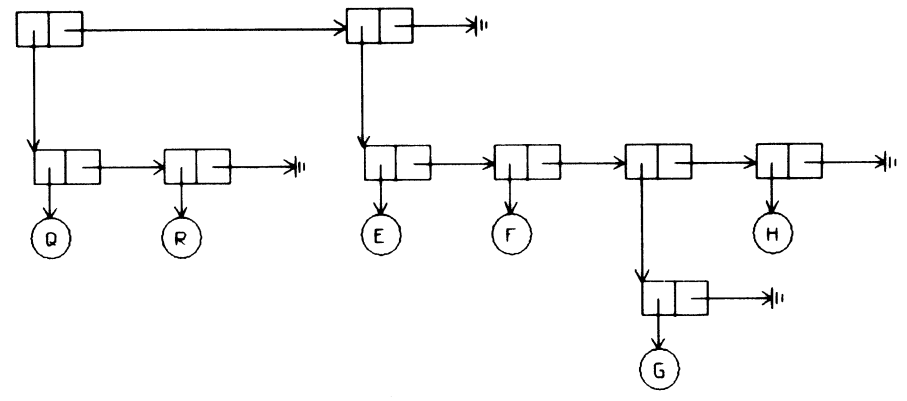
   b)



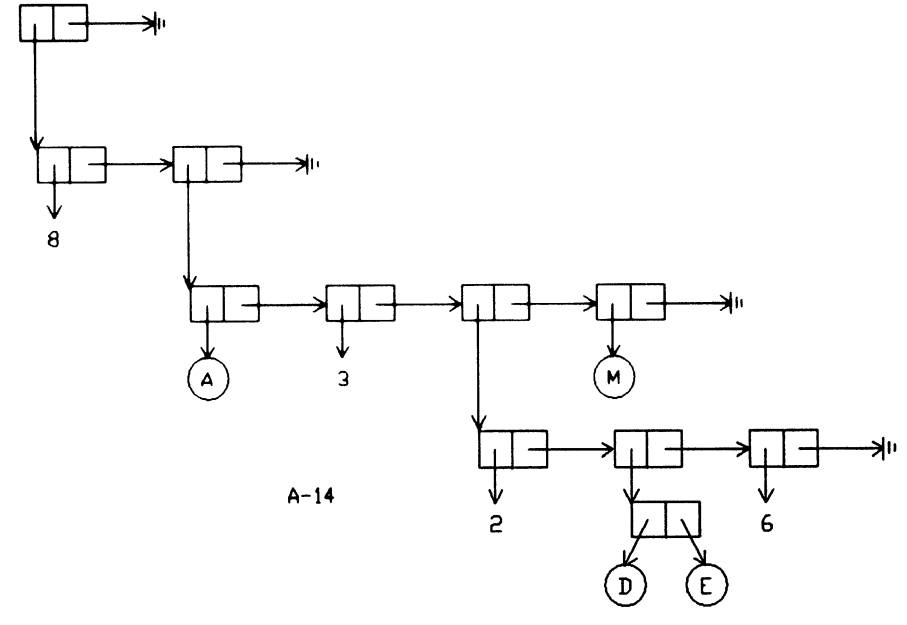
   c)



d)



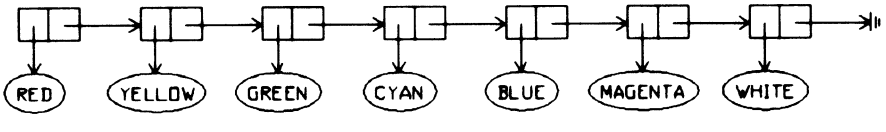
e)



A-14

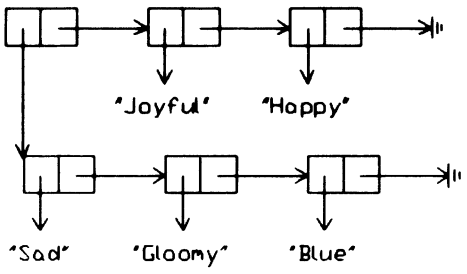
3. For each of the following lists, determine the car, the cdr, and the cadr, and the sequence of cars and cdrs that will return `blue`.

a)



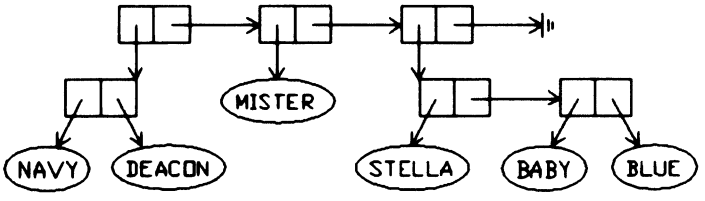
car = `red`
cdr = `(yellow green cyan blue magenta white)`
cadr = `yellow`
`blue` = car of the cdr of the cdr of the cdr of the cdr = car of the cddddr

b)



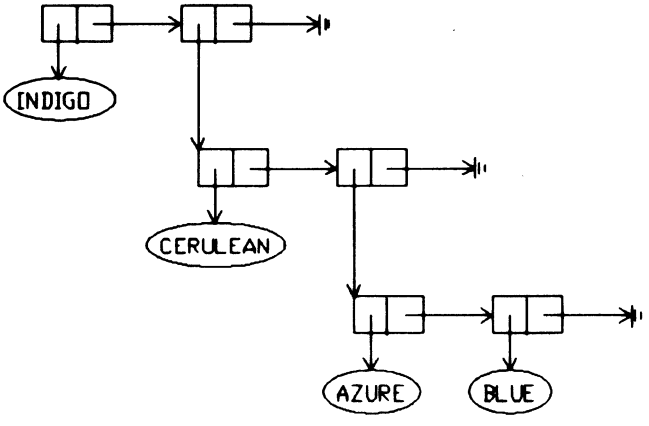
car = `("Sad" "Gloomy" "Blue")`
cdr = `("Joyful" "Happy")`
cadr = `"Joyful"`
`blue` = car of the cdr of the cdr of the car = caddar

c)



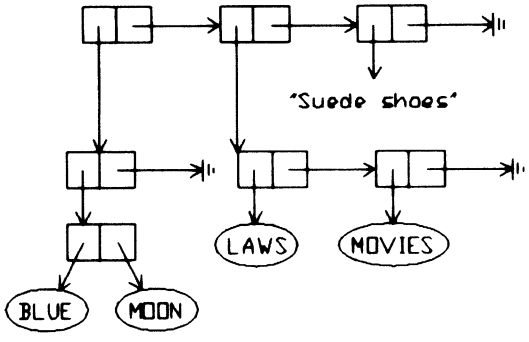
car = `(navy . deacon)`
cdr = `(mister (stella baby . blue))`
cadr = `mister`
`blue` = cdr of the cdr of the car of the cdr of the cdr = cdr of the cdaddr

d)



car = `indigo`
cdr = `((cerulean (azure blue)))`
cadr = `(cerulean (azure blue))`
`blue` = car of the cdr of the car of the cdr of the car of the cdr = cadr of the cadadr

e)



car = ((blue . moon))
cdr = ((laws movies) "Suede shoes")
cadr = (laws movies)
blue = car of the car of the car = caaar

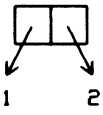
4. What does AutoLISP return when we enter each of these expressions?

```
Command: (setq list1 (list 1 2 3 4))
(1 2 3 4)

Command: (car list1)
1

Command: (cadr list1)
2

Command: (caddr list1)
3

Command: (cdr list1)
(2 3 4)

Command: (cddr list1)
(3 4)

Command: (cdddr list1)
(4)
```

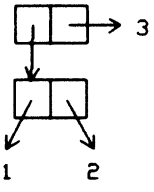
In this example, cars return numbers because these are the elements of the list. cdrs return lists because the cdr of a list is always a list. In fact, the cdr of a list is the remainder of that list if we remove the car.

5. Here is what's created when we run the specified functions:

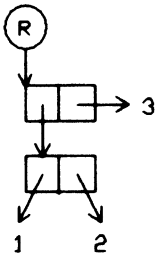
a)



b)

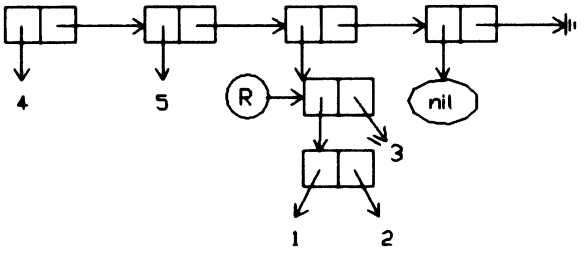


c)



The printed representation of this cons is `((1 . 2) . 3)`.

d)



The printed representation of this list is `(4 5 ((1 . 2) . 3) nil)`.

The length of this list is 4; there are four conses in its backbone. `nil` is an element of the list because it was supplied as an argument to the `list` function.

6. a)
```
(setq area (* pi r r))
```
or
```
(setq area (* pi (expt r 2)))
```

b)
```
(setq circum (* 2 pi radius))
```

c) `(setq r (sqrt (+ (* x x) (* y y) (* z z))))`

or

`(setq r (sqrt (+ (expt x 2) (expt y 2) (expt z 2))))`

d) `(setq d (/ (* w (expt 1 3)) (* 3 e i)))`

or

`(setq d (/ (* w 1 1 1) (* 3 e i)))`

e) `(setq p (/ (* w r (sin (+ phi beta))) a))`

Chapter 3. Arithmetic Functions

1. AutoLISP returns the following when we enter each of these expressions:

```
Command: (setq a (+ 2 3))          ;Numbers are added
5

Command: !a                        ;a is bound to the sum
5

Command: (setq b (* a 7))          ;Set b to the product of a multiplication
35

Command: (setq c (1+ b))           ;Add 1 to the value of b and assign to c
36

Command: !b                        ;The value of b is still 35
35

Command: (setq b (sqrt 1+ b))      ;We forgot a pair of parentheses.
error: bad argument type           ;1+ is an operand, not an operator.

Command: (setq b (sqrt (1+ b)))    ;Add 1 to b and find the square root
6.0

Command: !b                        ;Now b has been assigned a new value
6.0

Command: (setq d (fix b))          ;Convert b's value to an integer
6

Command: !b                        ;b is still a real number
6.0

Command: (setq e (/ c 2))          ;Set e to the quotient of a division
18

Command: (gcd c e)                 ;18 is largest number that divides
18                                 ;  evenly into 36 and 18

Command: (gcd b e)                 ;Both arguments must be integers
error: bad argument type

Command: (setq f (/ e 4))          ;Result of an integer division
4

Command: (setq g (/ e 4.0))        ;Result of a real division
4.5
```

```
Command: (setq h (/ (float e) 4))   ;Another way to do a real division
4.5

Command: (rem e 4)                  ;Remainder of (/ 18 4)
2

Command: (max a c e 3)              ;36 is largest number
36

Command: (max a b c e 3)            ;36 still largest, but b's value forces
36.0                                ; a real result

Command: (min a -8 f 6.0)           ;-8 is the most negative number.
-8.0                                ;6.0 forces a real result
```

2. The expressions `(lsh int1 int2)` and `(* int1 (expt 2 int2))` are equivalent. To see why, let's look at an example.

```
Command: (lsh 3 4)
48
```

The number 3_{10} is equal to 00000011_2. If we shift it 4 bits to the left we get 00110000_2, which is 48_{10}. Now let's execute the second expression:

```
Command: (* 3 (expt 2 4))
48
```

2^4 is equal to 16. When we multiply 16 by 3, we again get 48. Why does this happen?

Since bits are expressed in base 2, every time we shift left one bit we are multiplying the number by 2. In other words, the second argument, which specifies the number of bit positions to shift, also specifies the power to which the number 2 is raised.

Chapter 4. Evaluation of AutoLISP Expressions

1. What will each of the following return?

```
Enter Form: (+ 5 6)
11

Enter Form: (setq a 4)
4

Enter Form: (setq b a)
4

Enter Form: (setq c (+ 7 2))
9

Enter Form: (setq d (+ (a b)))
error: bad function

Enter Form: (setq d (+ a c))
13

Enter Form: (setq e (+ d (* a 3)))
25

Enter Form: (/ e a)
6

Enter Form: e
25

Enter Form: (/ e (float a))
6.25

Enter Form: (setq f (list a c e))
(4 9 25)

Enter Form: (f e e d)
error: bad formal argument list

Enter Form: (car f)
4

Enter Form: (car f f)
error: too many arguments

Enter Form: (cdr f)
(9 25)
```

```
Enter Form: (cadr f)
9

Enter Form: (x y z)
error: null function
```

2. a) `(1+ -7)`
operator: `1+`
operand: –7
argument: –7
result: –6

b) `(expt 7 2)`
operator: `expt`
operands: 7 and 2
arguments: 7 and 2
result: 49

c) `(/ x y)`
operator: `/`
operands: `x` and `y`
arguments: 4 and 5
result: 0

d) `(/ (float x) y)`
operator: `/`
operands: `(float x)` and `y`
2nd level operator: `float`
2nd level operand: `x`
2nd level argument: 4
2nd level result: 4.0
arguments: 4.0 and 5
result: 0.8

e) `(list x y z)`
operator: `list`
operands: `x`, `y`, and `z`
arguments: 4, 5, and `nil`
result: `(4 5 nil)`

f) `(+ x y z)`
operator: `+`
operands: `x`, `y`, and `z`
arguments: 4, 5, and `nil`
result: "bad argument type" error

g) `(cons x y z)`
operator: `cons`
operands: `x`, `y`, and `z`
arguments: 4, 5, and `nil`
result: "too many arguments" error

h) `(cons x y)`
operator: `cons`
operands: `x` and `y`
arguments: 4 and 5
result: `(4 . 5)`

i) `(* 4 (+ x (/ 21 3) 2))`
operator: `*`
operands: 4 and `(+ x (/ 21 3) 2)`
2nd level operator: `+`
2nd level operands: `x`, `(/ 21 3)`, and 2
3rd level operator: `/`
3rd level operands: 21 and 3
3rd level arguments: 21 and 3
3rd level result: 7
2nd level arguments: 4, 7, and 2
2nd level result: 13
arguments: 4 and 13
result: 52

j) `x`
result: 4
(Remember, anything can be a form. It's quite acceptable to give a symbol to `eval`.)

Chapter 5. Writing and Running Functions

1. a) `(quote x)`
   operator: `quote`
   operand: `x`
   result: `X`

   b) `(list x y)`
   operator: `list`
   operands: `x` and `y`
   arguments: 4 and 5
   result: `(4 5)`

   c) `(list x (quote y))`
   operator: `list`
   operands: `x` and `(quote y)`
   2nd level operator: `quote`
   2nd level operand: `y`
   2nd level result: `y`
   arguments: 4 and `y`
   result: `(4 Y)`

   d) `(list x 'y)`
   operator: `list`
   operands: `x` and `(quote y)`
   2nd level operator: `quote`
   2nd level operand: `y`
   2nd level result: `y`
   arguments: 4 and `y`
   result: `(4 Y)`

   e) `(+ x 'y)`
   operator: `+`
   operands: `x` and `(quote y)`
   2nd level operator: `quote`
   2nd level operand: `y`
   2nd level result: `y`
   arguments: 4 and `y`
   result: "bad argument type" error

   f) `(quote (x y))`
   operator: `quote`
   operands: `(x y)`
   result: `(X Y)`

g) `(quote x y)`
operator: `quote`
operands: `x` and `y`
result: "incorrect number of arguments" error

h) `(list '(x y))`
operator: `list`
operand: `(quote (x y))`
2nd level operator: `quote`
2nd level operand: `(x y)`
2nd level result: `(x y)`
argument: `(x y)`
result: `((X Y))`

i) `(list (x y))`
operator: `list`
operand: `(x y)`
2nd level operator: `x`
2nd level operand:
2nd level result: "bad function" error
arguments:
result:

Due to the error `eval` never sees the operand

j) `(list x '(y))`
operator: `list`
operands: `x` and `(quote (y))`
2nd level operator: `quote`
2nd level operand: `(y)`
2nd level result: `(y)`
arguments: 4 and `(y)`
result: `(4 (Y))`

k) `(setq x 'y)`
operator: `setq`
operands: `x` and `(quote y)`
2nd level operator: `quote`
2nd level operand: `y`
2nd level result: `y`
argument: `y`
result: `setq` binds `x` to `y` then returns `Y`.

`setq` is a special operator; its first operand is not evaluated. It runs on the operand `x` and the argument `y`.

l) `(setq a x b y)`
operator: `setq`
operands: `a`, `x`, `b`, and `y`
arguments: `y` and 5
result: `setq` binds `a` to `y` and `b` to 5, then returns 5.

The operands `a` and `b` are not evaluated. The operands `x` and `y` are evaluated, and their values become arguments.

m) `(setq c (x + y))`
operator: `setq`
operands: `c` and `(x + y)`
2nd level operator: `x`
2nd level operand:
2nd level result: "bad function" error

2. `roundit` rounds positive numbers.

```
(defun roundit (num)
  (fix (+ num 0.5)))
```

3. `rtd` converts radians to degrees.

```
(defun rtd (angle)
  (* (/ angle pi) 180.0))
```

Note that this function is a good candidate for our `acad.lsp` file.

4. `f-to-c` accepts a Fahrenheit temperature returns its Celsius equivalent.

```
(defun f-to-c (f)
  (* (/ (- f 32) 9.0) 5))
```

or

```
(defun f-to-c (f)
  (* (/ 9 5.0) (- f 32)))
```

5. `tip` takes a real number representing dollars, rounds it down to the nearest dollar, and calculates a 15 percent tip on this amount.

```
(defun tip (price)
  (* 0.15 (fix price)))
```

6. `add-tip` takes a real number representing dollars and returns this amount with a tip added on.

```
(defun add-tip (amount)
  (+ amount (tip amount)))
```

7. `ins-str` inserts the supplied string at the specified point in the drawing.

```
(defun ins-str (str pt)
  (command "text" pt "" "" str))
```

8. The following commands perform ZOOM EXTENTS, PREVIOUS, and WINDOW, respectively.

```
(defun c:ze ()
  (command "zoom" "e"))

(defun c:zp ()
  (command "zoom" "p"))

(defun c:zw ()
  (command "zoom" "w" pause pause))
```

9. Here is one approach to writing a function that takes a list as its only argument and returns the third element of that list.

   The `third` function requires one argument, a list. The result it returns can be any type of AutoLISP object . . . whatever the third element of the supplied list happens to be. Therefore, if the user supplies the 3D point, `(2.0 3.3 4.5)`, it returns the Z coordinate, 4.5. If the user supplies a list of strings, `("dsk-secy" "dsk-mgr" "chr-secy" "chr-mgr")`, it returns the string, `"chr-secy"`.

   We start every function we write with `(defun`, followed by the name of the function. This function is called `third`, but you can call it `ray`, or just about anything you want. However, `third` is a good choice because it conveys information about the function's contract.

   Next comes the parameter list. `third` takes one argument and therefore must have one parameter. Let's call the parameter `lst`, because our expected argument is a list. Remember, if we call the parameter `list` we will lose access to the `list` function. Our first line is complete and looks like this:

```
(defun third (lst)
```

   Next comes the body of our function; in this case it's only one line long. Remembering our cars and cdrs (we hope!), the `third` element of any list is the car of the cdr of the cdr of that list. Thus, the second line runs these three functions together:

```
(car (cdr (cdr lst)))
```

Had the user entered (third '(a b c d e)), then the supplied list would be the value of lst. Our code first finds the cdr of lst, which is (b c d e). A second cdr gets (c d e). It takes the car and, voilà!, it returns c, the third element of the list.

The entire function, with proper indentation and a close parenthesis to match the open parenthesis before the defun, looks like this:

```
(defun third (lst)
  (car (cdr (cdr lst))))
```

We can optionally shorten this to the following:

```
(defun third (lst)
  (caddr lst))
```

10. kill-3 takes a list as its only argument and returns a portion of that list minus the first three elements.

```
(defun kill-3 (l)
  (cdr (cdr (cdr l))))
```

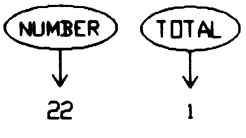
11. midpt takes two 2D points and returns the midpoint between them.

```
(defun midpt (x y)
  (list (/ (+ (car x) (car y)) 2.0)
        (/ (+ (cadr x) (cadr y)) 2.0)))
```

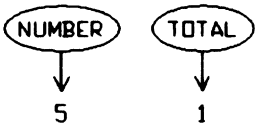
12. c:endit uses WBLOCK to save the drawing, then exits. The first time the drawing is saved it returns the message "Block YES not found" but still runs correctly. (c:endit is based on a command by Ray White.)

```
(defun c:endit (/ name)
  (setvar "cmdecho" 0)                    ;Turn off echoing
  (setq name (getvar "dwgname"))          ;Get the drawing name
  (command "wblock" name "yes" "*")       ;Write the drawing
  (command "quit" "yes"))                 ;Quit the drawing
```

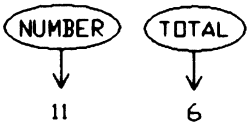
13. Prior to entering test, the value of number and total are 22 and 1:



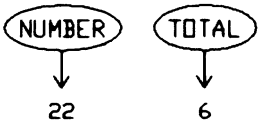
Since number is a parameter, it gets as its value the argument, 5. total retains its global value:



The test function sets total to 6, which is the sum of total (1) + number (5). It then sets number to 11, which is the sum of number (5) + total (now 6):



Finally, test prints the value of total (6), number (11), and count (nil, because count is unbound), then returns count's value. When the function is finished, number is rebound to its global value (22) and total retains its new value (6):



14. If we initialize m to 5 and n to 6, what gets printed by subsequent calls to p and v?

```
;;;m and n have global values.
(defun p ()
  (print m)                                ;5 - global value of m
  (print n)                                ;6 - global value of n
  (q 9))                                   ;Call q, pass it 9

;;;m is bound to the value passed as argument, n uses its local value.
(defun q (m / n)
  (print m)                                ;9 - passed as a parameter
  (print n)                                ;nil - n is unbound
  (setq m 22)                              ;Rebind m
  (r 4))                                   ;Call r, pass it 4

;;;m keeps value assigned in q. n's value is passed as an argument.
(defun r (n)
  (print m)                                ;22 - assigned in q
  (print n)                                ;4 - passed as a parameter
  (s))                                     ;Call s
```

```
;;;m and n are not defined locally. They keep previous values.
(defun s ()
  (print m)                                ;22 - assigned in q
  (print n))                               ;4 - assigned in r
                                           ;4 is returned

;;;We call v from the Command: prompt. m and n are global.
(defun v ()
  (print m)                                ;5 - assigned globally
  (print n))                               ;6 - assigned globally
                                           ;6 is returned
```

Chapter 6. Control Structure: Producing Powerful Programs

1.
```
Command: (numberp 5.0)
T

Command: (zerop 5.0)
nil

Command: (minusp 5.0)
nil

Command: (type 5.0)
REAL

Command: (type b)
LIST

Command: (type (car a))
INT

Command: (type (car b))
INT

Command: (type (car c))
SYM

Command: (eq a b)
nil

Command: (eq b d)
T

Command: (equal a b)
T

Command: (equal a c)
nil

Command: (eq a a)
T

Command: (equal x (cadr b) 1.1)
T

Command: (eq "abc" "abc")
T

Command: (eq "abc" "ABC")
nil
```

2. not and null do the exact same thing: They both return t when handed nil, and return nil when given anything else. They are applied differently, however. not is used to reverse the result of a test to if or while. null is used to specifically test for the symbol nil.

3. evenp returns t if its argument is even, and nil otherwise.

```
(defun evenp (n)
  (if (zerop (rem n 2))                    ;Is remainder of n/2 = 0?
      t                                    ;Yes, n must be even
      nil))
```

There's actually an even easier way to write this function. Since the if returns exactly what zerop returns, it isn't needed:

```
(defun evenp2 (n)
  (zerop (rem n 2)))                       ;Is remainder of n/2 = 0?
```

4. symbolp returns t if its argument is a symbol and nil otherwise.

```
(defun symbolp (object)
  (if (eq (type object) 'sym)              ;Is object of type symbol?
      t                                    ;Yes, return T
      nil))                                ;No, return nil
```

or

```
(defun symbolp2 (object)
  (eq (type object) 'sym))
```

Both versions of this function have a minor bug. The type function returns nil if it is supplied with the symbol nil. Thus if we give nil to symbolp, it will return nil, even though nil is a symbol! The following code corrects this problem:

```
(defun symbolp3 (object)
  (if (or (null object)                    ;Is argument either nil
          (equal (type object) 'sym))      ;  or a symbol?
      t                                    ;Yes
      nil))                                ;No
```

5. ceiling takes a real number and "rounds" it up to the next higher integer.

```
(defun ceiling (n)
  (if (= n (fix n))                        ;Already a whole number?
      (fix n)                              ;Yes, convert to integer
      (if (minusp n)
          (fix (1- n))                     ;If negative, subtract one
          (fix (1+ n)))))                  ;Else add one
```

6. The grades function analyzes a grade:

```
(defun grades (grade)
  (cond ((< 89 grade) 'excellent)
        ((< 79 grade 90) 'good)
        ((< 64 grade 80) 'not-so-good)
        ((< grade 65) 'failing)))
```

If your version doesn't work, you may have made one of the following errors:

- cond requires parens in very specific places. Make sure that yours are situated correctly.
- < means "less than"; don't confuse it with >, which is "greater than."
- Be sure that you tested the boundary cases: 65, 80, and so on.
- Did you write the symbol not-so-good without hyphens? If so, AutoLISP evaluates three separate symbols: not, so, and good. The value of the last evaluated symbol, good, is probably nil, so cond returns nil for that range.

Here's the same function with an if instead of a cond:

```
(defun grades2 (grade)
  (if (< 89 grade)
      'excellent
      (if (< 79 grade 90)
          'good
          (if (< 64 grade 80)
              'not-so-good
              'failing))))
```

7. zm takes the string "w", "p", or "e" and performs a ZOOM WINDOW, PREVIOUS, or EXTENTS.

```
(defun zm (str)
  (cond
    ((or (= str "w") (= str "W")) (command "zoom" "w" pause pause))
    ((or (= str "p") (= str "P")) (command "zoom" "p"))
    ((or (= str "e") (= str "E")) (command "zoom" "e"))
    (t "Sorry, not an accepted zoom type.")))
```

8. expon verifies that both arguments are numbers and that the second is between 2 and 4, then raises the first number to the power specified by the second.

```
(defun expon (number power)
  (if (and (numberp number) (numberp power) (< 1 power 5))
      (expt number power)
      'error))
```

If we called this function expt, we'd rebind the symbol expt and lose access to the exponentiation subr. Be careful!

9. nthcdr takes a number and a list, and "cdrs" the list *number* times.

```
(defun nthcdr (n l)
  (repeat n
    (setq l (cdr l))))
```

10. third-5 prints the third element of a list five times.

```
(defun third-5 (l)
  (repeat 5
    (print (caddr l))))
```

11. multby2 takes a list of numbers and prints each number multiplied by 2.

```
(defun multby2 (l)
  (foreach num l
    (print (* num 2))))
```

12. non-num prints each list element prior to the first non-number, followed by the count of numbers it printed. We use while instead of foreach because we may not want to access all of the elements in the list.

```
(defun non-num (l / total)
  (setq total 0)                              ;Initialize counter
  (while (numberp (car l))                    ;While next element is a number
    (print (car l))                           ;Print it
    (setq l (cdr l))                          ;Advance list to next cons
    (setq total (1+ total)))                  ;Increment counter
  (print "Numbers at the start of the list: ") ;When non-number encountered,
  total)                                      ;  end loop and return total
```

13. sum and sum2 return the sum of a list of numbers.

```
(defun sum (list1 / total)
  (setq total 0)
  (foreach item list1
    (setq total (+ total item))))
```

```
(defun sum2 (x)
  (if (null x)                          ;At end of list?
      0                                 ;Yes, return 0
      (+ (sum (cdr x)) (car x))))       ;No, call ourselves with cdr and add car
```

14. numsp and o-num test whether there are only numbers in a list.

```
(defun numsp (list1 / flag)
  (setq flag t)                         ;Initialize flag
  (foreach item list1                   ;Check each list element
    (if (not (numberp item))            ;If any elt isn't a number,
        (setq flag nil)))               ;  set flag to nil
  flag)                                 ;Return flag when done

(defun o-num (list1)
  (if (null list1)                      ;If end of list
      t                                 ;  return t.  It's all numbers.
      (if (numberp (car list1))         ;Otherwise, is car a number?
          (o-num (cdr list1))           ;Yes, call ourselves with rest of list
          nil)))                        ;No, return nil
```

15. biggest and largest return the largest of a list of numbers.

```
(defun biggest (list1 / big)
  (setq big (car list1))                ;Initialize big to first element
  (foreach elt list1
    (if (> elt big)                     ;Is next element bigger than big?
        (setq big elt)))                ;Yes, replace it
  big)                                  ;Return biggest number

(defun largest (list1)
  (if (null (cdr list1))                ;One-element list?
      (car list1)                       ;Yes, return the car
      (if (> (car list1) (largest (cdr list1))) ;Is car larger than largest of cdr?
          (car list1)                   ;Yes, return car
          (largest (cdr list1)))))      ;No, check out cdr
```

16. last-cons and last-cons2 return the last cons in a list.

```
(defun last-cons (list1)
  (while (cdr list1)                    ;While cdr of list1 is not nil
    (setq list1 (cdr list1)))           ;  get next backbone cons
  list1)                                ;Finally, return last cons
```

In last-cons, the body of the while loop repeatedly binds list1 to the next cons. On the last pass through the loop, the last cons is propagated back through the setq and while and is returned by the defun. In other words, the last form, list1, is almost unnecessary. It is there to handle the specific instance where the function is given a one-element list:

`(last-cons '(a))`. Try running the function with and without the `list1` at the end. Give it a one-element list and see what happens.

```
(defun last-cons2 (list1)
  (if (null (cdr list1))          ;At end of list?
      list1                       ;Yes, return this cons
      (last-cons (cdr list1))))   ;No, get cdr and try again
```

17. `zi` repeatedly prompts the user for a point and performs a ZOOM CENTER at half the VIEWSIZE.

```
(defun zi (/ pt size)
  (setq pt (getpoint "\nSelect ZOOM IN point: "))
  (while pt
    (setq size (/ (getvar "viewsize") 2))
    (command "zoom" "c" pt size)
    (setq pt (getpoint "\nSelect ZOOM IN point: "))))
```

`zo` repeatedly prompts the user for a point and performs a ZOOM CENTER at twice the VIEWSIZE. Note the different use of `setq` from `zi`.

```
(defun zo (/ pt size)
  (while (setq pt (getpoint "\nSelect ZOOM OUT point: "))
    (setq size (* (getvar "viewsize") 2))
    (command "zoom" "c" pt size)
    (setq pt (getpoint "\nSelect ZOOM OUT point: "))))
```

Add `"i"` and `"o"` options to the `cond` you created in Lab Exercise 7.

```
(defun zz (str / pt size)
  (cond
    ((or (= str "w") (= str "W")) (command "zoom" "w" pause pause))
    ((or (= str "p") (= str "P")) (command "zoom" "p"))
    ((or (= str "e") (= str "E")) (command "zoom" "e"))
    ((or (= str "i") (= str "I")) (zi))
    ((or (= str "o") (= str "O")) (zo))
    (t (princ "Sorry, not an accepted zoom type."))))
```

18. `every-other` and `e-o` print every other element in a list.

```
(defun every-other (l)
  (repeat (/ (length l) 2)    ;Do n/2 times
    (print (car l))           ;Print first element
    (setq l (cddr l)))        ;Skip 2 elements
  (if l                       ;If odd # of elts, value of l not nil
    (print (car l))))         ;  so print last elt
```

```
(defun e-o (l / count)
  (setq count 1)
  (foreach elt l
    (if (not (evenp count))    ;We wrote evenp in Exercise 3
        (print elt))           ;If count is even, print element
    (setq count (1+ count))))  ;Note that count is returned
```

19. `put-strs` takes a list of strings and a list of points and places each string at the corresponding point in the drawing.

```
(defun put-strs (sl pl)
  (if (not (= (length sl) (length pl)))          ;Here's an optional test. length
      (princ "\nLists must be of equal length")  ;  function shown in next chapter
      (foreach pt pl
        (command "text" pt "" "" (car sl))       ;TEXT command will do it
        (setq sl (cdr sl)))))                    ;Get next string
```

20. `primep` determines whether a given number is prime. A number is prime if it is indivisible by a number between 2 and its square root. For example, when testing whether 26 is prime, there's no need to test beyond 5. Anything larger would have to be multiplied by a number that is smaller than 5, and we've already checked those possibilities.

```
(defun primep (x / count flag)
  (setq count 2
        flag  t)
  (if (/= x 2)                                   ;We know 2 is prime
      (repeat (fix (sqrt x))                     ;Only need to test to sqrt
        (if (zerop (rem x count))                ;If x is divisible by count
            (setq flag nil))                     ;  then not prime. Set flag
        (setq count (1+ count))))                ;Check next number
  flag)                                          ;flag has t or nil.
```

21. `prime-print` prints all primes up to the given number.

```
  (defun prime-print (x / count)
    (setq count 1)
    (repeat x
      (if (primep count)                         ;Is current number prime?
          (print count))                         ;Yes, print it
      (setq count (1+ count)))
    (prin1))
```

Chapter 7. Working with Lists

1. a) Create the list (MA NY GA WA CA).

```
Command: (setq sa '(ma ny ga wa ca))
(MA NY GA WA CA)
```

or

```
Command: (setq sa (list 'ma 'ny 'ga 'wa 'ca))
(MA NY GA WA CA)
```

b) Remove Massachusetts from the list.

```
Command: (setq sb (cdr sa))
(NY GA WA CA)
```

c) Remove California from the list.

```
Command: (setq sc1 (reverse sb))
(CA WA GA NY)

Command: (setq sc2 (cdr sc1))
(WA GA NY)

Command: (setq sc (reverse sc2))
(NY GA WA)
```

or

```
Command: (setq sc (reverse (cdr (reverse sb))))
(NY GA WA)
```

d) Add Ohio to the front of the list.

```
Command: (setq sd (cons 'oh sc))
(OH NY GA WA)
```

e) Add Ohio to the end of the list.

```
Command: (setq se (reverse (cons 'oh (reverse sd))))
(OH NY GA WA OH)
```

or

```
Command: (setq se (append sd '(oh)))
(OH NY GA WA OH)
```

f) Copy the last element of `se` to the front of the list.

```
Command: (setq sf (cons (last se) se))
(OH OH NY GA WA OH)
```

g) Create another list `(bc ont que)`.

```
Command: (setq sg '(bc ont que))
(BC ONT QUE)
```

h) Merge `sf` and `sg` into a single new list.

```
Command: (setq sh (append sf sg))
(OH OH NY GA WA OH BC ONT QUE)
```

i) Copy the seventh element of `sh` to the front of `sh`.

```
Command: (setq si (cons (nth 6 sh) sh))
(BC OH OH NY GA WA OH BC ONT QUE)
```

j) Replace all occurrences of Ohio with Texas.

```
Command: (setq sj (subst 'tx 'oh si))
(BC TX TX NY GA WA TX BC ONT QUE)
```

k) Return the portion of `sj` beginning with Georgia.

```
Command: (setq sk (member 'ga sj))
(GA WA TX BC ONT QUE)
```

2. `add-end` adds an object to the end of a list.

```
(defun add-end (list1 elt)
  (append list1 (list elt)))
```

or

```
(defun add-end2 (list1 elt)
  (reverse (cons elt (reverse list1))))
```

3. `add-pair` adds a pair to the front of an association list.

```
(defun add-pair (pair alist)
  (cons pair alist))
```

4. **Command:** (setq al (subst '(a . "jkl") '(a . "xyz") al))
((A . "jkl") (B . 1.23457) (C 7.65432 9.87679))

Command: (setq al (subst '(b . 1.3) (assoc 'b al) al))
((A . "jkl") (B . 1.3) (C 7.65432 9.87679))

Command: (setq al (subst '(c 7.0 9.0) (assoc 'c al) al))
((A . "jkl") (B . 1.3) (C 7.0 9.0))

5. `remove` returns a copy of the list with all occurrences of the supplied object removed.

```
(defun remove (obj list1 / list2)
  (foreach elt list1
    (if (not (equal elt obj))            ;If obj does NOT match element,
        (setq list2 (cons elt list2))))  ;  cons it onto list
  (reverse list2))
```

Note: Since we cons onto the front of a list, we reverse the list when we're done in order to have it returned in the correct order. This technique is employed in numerous functions we'll study in this book.

6. `count`, `count2`, and `count3` return the number of times an object is a member of a list.

```
(defun count (obj l / num)
  (setq num 0)
  (foreach elt l
    (if (equal obj elt)                  ;If obj matches element,
        (setq num (1+ num))))            ;  increment counter
  num)                                   ;Return counter

(defun count2 (elt l / cnt)
  (setq cnt 0)
  (while (setq l (member elt l))         ;While elt is a member, get sublist
    (setq l (cdr l))                     ;  cdr past it,
    (setq cnt (1+ cnt)))                 ;  and increment count
  cnt)                                   ;Return count when done.

(defun count3 (obj l / num)
  (if (null l)                           ;If end of list
      0                                  ;  return 0. Else,
      (+ (if (equal obj (car l))         ;Does obj match car of list?
             (1+ (count3 obj (cdr l)))   ;Yes: Add 1 and call self w/cdr
             (count3 obj (cdr l))))))    ;No: Don't add 1, call self w/cdr
```

7. `multby` and `multbyn` take a number and a list of numbers and multiply each element of the list by that number.

```
(defun multby (n l1 / l2)
  (foreach elt (reverse l1)                    ;Reverse so it's forward when we cons on
    (setq l2 (cons (* elt n) l2))))

(defun multbyn (n l)
  (mapcar '(lambda (elt)                       ;Map a function
             (* elt n))                        ;  to perform multiplication
          l))                                  ;  on a supplied list.
```

8. `cop` and `cop2` return a copy of a list.

```
(defun cop (list1 / list2)
  (foreach elt list1
    (setq list2 (cons elt list2)))             ;Cons each elt onto front of list
  (reverse list2))

(defun cop2 (list1)
  (if (null list1)                             ;End of list?
      nil                                      ;Yes, return nil
      (cons (car list1) (cop2 (cdr list1)))))  ;No, cons car onto remainder
```

Another solution is to simply enter `(append list1)`. `append` creates a copy of its argument.

9. `evens` and `evens2` take a list of integers and return a list of the even ones.

```
(defun evens (l1 / l2)
  (foreach elt l1
    (if (evenp elt)                            ;We wrote evenp in Chapter 6 lab
        (setq l2 (cons elt l2))))              ;If even, cons onto list
  (reverse l2))

(defun evens2 (l1 / l2)
  (mapcar '(lambda (elt)
             (if (evenp elt)                   ;If element is even number
                 (setq l2 (cons elt l2))))     ;  cons it onto list
          l1)
  (reverse l2))                                ;Reverse and return list
```

10. `other` and `other2` build a list of every other member of the supplied list.

```
(defun other (l1 / l2)
  (repeat (/ (length l1) 2)                    ;Do n/2 times
    (setq l2 (cons (car l1) l2))               ;Add first element
    (setq l1 (cddr l1)))                       ;Skip 2 elements
  (if l1                                       ;If odd # of elts, value of l not nil
     (setq l2 (cons (car l1) l2)))             ;Add last element
  (reverse l2))

(defun other2 (list1)
  (if (null list1)                             ;End of list?
      nil                                      ;Yes, return nil. If not, call self
      (cons (car list1) (other2 (cddr list1))))) ;  with cddr and cons car onto front
```

11. `assoc2` works like `assoc` except that it keys on the cdr of the sublist rather than the car.

```
(defun assoc2 (obj lst / result)
  (foreach pair (reverse lst)                 ;Examine list in reverse order
    (if (eq (cdr pair) obj)                   ;If obj matches cdr of pair
        (setq result pair)))                  ;  assign it to result, then keep going
  result)                                     ;Return last one found
```

12. `map-al` takes two lists of arbitrary length and creates an association list.

```
(defun map-al (list1 list2)
  (mapcar 'cons list1 list2))
```

13. These are alternatives for summing a list. The former isn't considered the best AutoLISP programming style, but it's interesting to consider.

```
(defun sum3 (list1)
  (setq list1 (cons '+ list1))     ;Make + the car of the list
  (eval list1))                    ;  then evaluate the list

(defun sum4 (list1)
  (apply '+ list1))
```

14. `put-strs2` is the same as `put-strs` except that it uses `lambda`.

```
(defun put-strs2 (sl pl)
  (mapcar '(lambda (sl1 pl1)
             (command "text" pl1 "" "" sl1))
          sl pl))
```

15. connect takes a point list and a string and draws either an open line or a closed pline connecting the specified points. The function conses the information onto the front of the list, then reverses the list at the end. It forces evaluation of the command function by explicitly calling eval.

```
(defun connect (l obj / c comm)
  (if (or (equal obj "line")                    ;Test upper & lower case
          (equal obj "LINE"))                   ;User wants line or pline?
      (setq c "")                               ;LINE: make it open
      (setq c "c"))                             ;PLINE: make it closed.
  (setq comm (list obj 'command))               ;Start to build command in reverse order
  (foreach pt l                                 ;Get each point in list of points
    (setq pt (list 'quote pt)                   ;Quote the point: (quote (1 1))
        comm (cons pt comm)))                   ;  and cons it onto command list
  (setq comm (cons c comm))                     ;Indicate "" or "c" at front of list
  (eval (reverse comm)))                        ;Reverse list and evaluate it
```

It's not really considered elegant to call eval explicitly when we can avoid it. The conn function does the same thing as connect but uses apply to simplify the process.

```
(defun conn (l obj / c comm)
  (if (or (equal obj "line")                    ;Test upper & lower case
          (equal obj "LINE"))                   ;User wants line or pline?
      (setq c "")                               ;LINE: make it open
      (setq c "c"))                             ;PLINE: make it closed.
  (setq l (append (list obj) l (list c)))       ;List of obj, points, and c
  (apply 'command l))                           ;Apply the command function to the list
```

16. conn2 does the same as the previous exercise but sets the layer as well.

```
(defun conn2 (l obj / c comm)
  (if (or (equal obj "line")                    ;Test upper & lower case
          (equal obj "LINE"))                   ;User wants line?
      (progn (setq c "")                        ;Yes, make it open
             (command "layer" "M" 0 ""))        ;  and place on layer 0
      (progn (setq c "c")                       ;No, pline; make it closed
             (command "layer" "M" 1 "")))       ;  and place on layer 1
  (setq l (append (list obj) l (list c)))       ;List of obj, points, and c
  (apply 'command l))                           ;Apply the command function to the list
```

Chapter 8. Input/Output

1. Convert the `grades` function into a command.

```
(defun c:grades (/ grade)
  (initget 7)
  (setq grade (getint "\nWhat is your grade? "))
  (cond ((< 89 grade) 'excellent)
        ((< 79 grade 90) 'good)
        ((< 64 grade 80) 'not-so-good)
        ((< grade 65) 'failing)))
```

or

```
(defun c:grades2 (/ grade)
  (initget 7)
  (setq grade (getint "\nWhat is your grade? "))    ;We can call the grades function
  (grades grade))                                   ;  we already wrote
```

2. Transform the `zz` function into a command. (`c:zz` and its component functions are based on a command by Jim Porter.)

```
(defun c:zz (/ str)
  (initget "W P E I O")
  (while (setq str (getkword "\nWindow/Previous/Extents/In/Out: "))
    (cond
      ((= str "W") (command "zoom" "w" pause pause))
      ((= str "P") (command "zoom" "p"))
      ((= str "E") (command "zoom" "e"))
      ((= str "I") (zi))
      ((= str "O") (zo)))
    (initget "W P E I O")))      ;Must reissue initget each pass
```

Note that `getkword` handles the check for both uppercase versus lowercase input and illegal input, which greatly simplifies this command.

3. `y-np` and `y-np2` are used to ask the user a yes or no question.

```
(defun y-np (string / ans)
  (initget "Yes No")
  (setq ans (getkword (strcat string " (Yes or <No>) ")))
  (if (eq ans "Yes")
      t
      nil))
```

```
(defun y-np2 (string)
  (initget "Yes No")                                          ;equal returns t or nil.
  (equal (getkword (strcat string " (Yes or <No>) ")) "Yes")) ;Don't need if.
```

When writing AutoLISP programs, we often need to ask the operator a yes or no question, then run code conditionally based on the response we receive. This function is a utility we can use in our own programs. For example, we might use it as a test in an `if`:

```
(if (y-np "Beam wide enough?")
    (princ "Wide enough")
    (princ "Not wide enough"))
```

It would appear to the user as follows:

```
Beam wide enough? (Yes or <No>) y
Wide enough
```

We will use the `y-np` function in several places throughout this book.

4. To replace the END command we need to do the following steps:

   a) Rename `c:endit` to `c:end`. Put it in the `acad.lsp` file.

   b) Place the following `s::startup` function in `acad.lsp`:

```
(defun s::startup ()
  (command "undefine" "end"))
```

5. `otos` converts any AutoLISP object to a string. It writes to a file named `"$"`.

```
(defun otos (obj / file)
  (setq file (open "$" "w"))              ;Open file named $
  (princ obj file)                        ;Write obj to file
  (setq file (close file))                ; as a string
  (setq file (open "$" "r"))              ;Reopen file to read
  (setq obj (read-line file))             ;Read string back in
  (setq file (close file))
  obj)                                    ;Return string
```

6. `c:parts` repeatedly prompts for a part number and part name pair until the user presses <enter>, then conses each pair onto an association list. When the user is done inputting data, this command writes the list out to a file.

```
(defun c:parts (/ pnum pname lst f)
  (setq pnum (getint "\nPart number (or <enter> when done): "))
  (while pnum                                          ;Until user hits <enter>
    (setq pname (read (getstring "\nPart name: "))     ;  get name and convert to symbol
          lst   (cons (cons pnum pname) lst)           ;Create pair and cons onto list
          pnum  (getint "\n\nPart number: ")))         ;Get next part number
  (setq f (open (getstring "Filename: ") "w"))         ;When done, open file,
  (prin1 lst f)                                        ;  write list to file,
  (setq f (close f))                                   ;  then close file
  (prin1))
```

7. `c:get-nums` reads in the list that was created in `c:parts`. It runs the `assoc2` function on each part name, and prints its associated part number.

```
(defun c:get-nums (/ f lst pname)
  (setq f (open (getstring "Filename: ") "r"))           ;Open file,
  (setq lst (read (read-line f)))                        ;  read data, convert to list,
  (setq f (close f))                                     ;  then immediately close file
  (setq pname (read                                      ;Get part name & convert
        (getstring "\nPart name (or CR when done): ")))
  (while pname                                           ;Until user hits <enter>
    (princ "\nThe part number for ")
    (prin1 pname)                                        ;  print part name
    (princ " is ")
    (prin1 (car (assoc2 pname lst)))                     ;  and part number
    (setq pname (read (getstring "\n\nPart name: "))))   ;Get next part name in loop
  (prin1))
```

8. The steps needed to perform simple file I/O is shown with the above solutions. To have `parts` request a filename, default the extension to `.dat`, and check for the file's existence, we can replace the code at the end with the following:

```
(while
  (setq f (open                                          ;Open file to see if it exists
     (setq name (strcat (getstring "Filename: ") ".dat")) "r"))
  (setq f (close f))                                     ;While it does, close it
  (prompt "\nFile already exists; supply another name.\n") ;  and request another
)
(setq f (open name "w"))                                 ;Finally, a new file! Open it,
(prin1 lst f)                                            ;  write data to it,
(setq f (close f))                                       ;  and close it
```

This code prompts for a filename and opens that file to read. If `open` does not return `nil`, then the file exists, so it closes that file and reprompts. When `open` finally returns `nil`, it opens this new file to write. This is a time-consuming algorithm because file access is comparatively slow.

An alternate solution is to search for the file with `findfile`:

```
(while (findfile (setq name (strcat (getstring "Filename: ") ".dat")))
  (prompt "\nFile already exists; supply another name.\n"))
```

A problem with `findfile` is that it might locate a file defined in a different directory along the AutoCAD library path.

Perhaps the best solution is to use `getfiled` with a flag value of 1. This restricts the search to the current directory and warns the user if an existing file is chosen.

```
(setq f (open (getfiled "Enter file to write" "" "dat" 1) "w"))
```

To have `get-nums` open a file to read we again run `getfiled`:

```
(setq f (open (getfiled "Select a data file to read" "" "dat" 4) "r"))
```

Here it prompts for a file with a `.dat` extension, but flag bit value 4 allows the user to enter a filename with a different extension.

Chapter 9. Expanding Our AutoLISP Toolbox

1. `str-rev` returns a copy of a string with the characters in reverse order. This algorithm starts with the last character in the string and adds it to the end of the new string, causing the new string to be built in reverse order.

```
(defun str-rev (s1 / count s2)
  (setq count (strlen s1)
        s2 "")                                    ;Initialize
  (while (> count 0)                              ;For each char starting at end
    (setq s2    (strcat s2 (substr s1 count 1))   ;Add char to end of string
          count (1- count))                       ;Dec. count to get previous char
    s2))                                          ;Return string
```

2. `dss` (***D***elete ***S***paces at ***S***tart) returns a string minus the leading spaces.

```
(defun dss (str / cnt)
  (setq cnt 1)
  (while (and (= (substr str cnt 1) " ")          ;While next char is space
              (< cnt (strlen str)))               ;  and not at end of string
    (setq cnt (1+ cnt)))                          ;Increment cnt
  (substr str cnt))                               ;Return substring starting at cnt
```

`dse` returns a string minus the trailing spaces. It uses the `str-rev` function from Lab Exercise 1.

```
(defun dse (str)
  (setq str (str-rev str))      ;Reverse string
  (setq str (dss str))          ;Remove spaces at start
  (str-rev str))                ;Reverse back
```

`ds` returns a string minus the leading and trailing spaces.

```
(defun ds (str)
  (dse (dss str)))
```

3. `c:sparse2` replaces strings containing numbers with the numbers themselves. It calls the `conv` function to perform the conversion. When we repeat a task it is often clearer to extract the code and make it its own function.

```
(defun c:sparse2 (/ s char cnt str-list word)
  (setq word ""
        cnt  0
        s    (getstring t "\nInput string: "))
  (while (< cnt (strlen s))                          ;If not end of string
    (setq cnt  (1+ cnt)                              ;  increment cnt and
          char (substr s cnt 1))                     ;  get next character
    (if (eq char " ")                                ;End of word?
        (if (not (eq word ""))                       ;Yes, if not a null string
            (setq str-list (cons (CONV WORD) str-list)    ; convert, cons onto str-list
                  word     ""))                      ;  and reset word
        (setq word (strcat word char)))              ;No, add char to end of word
  ) ;end of while
  (if (eq word "")                                   ;Any remaining characters?
      (reverse str-list)                             ;No, reverse and return list
      (reverse (cons (CONV WORD) str-list))))        ;Yes, convert to #, cons onto str-lst

;;;conv takes a string representing a number and converts it to the number
(defun conv (str)
  (if (numberp (read str))                           ;String a number?
      (read str)                                     ;Yes, convert it
      str))                                          ;No, return it as is
```

4. `c:stamp` puts a date and time stamp in a drawing.

```
(defun c:stamp (/ date yr mo da hour min sec)
  (setq date (rtos (getvar "cdate") 2 6)             ;Get date & time as decimal string
        yr   (substr date 3 2)                       ;Break it down into its components
        mo   (substr date 5 2)
        da   (substr date 7 2)
        hour (substr date 10 2)
        min  (substr date 12 2)
        sec  (substr date 14 2))
  (prompt "Start point: ")
  (command "text" pause "" ""                        ;Put in drawing formatted nicely
    (strcat mo "/" da "/" yr " " hour ":" min ":" sec))
  (prin1))
```

5. `zip` checks whether a string is a valid zip code. It calls the `check` function to see if the code is a number.

```
(defun zip (code)
  (cond ((= (strlen code) 5)                ;If 5 chars
         (check code))                      ;  test it
        ((= (strlen code) 10)               ;If 10 chars
         (and
           (check (substr code 1 5))        ;  test first 5
           (check (substr code 7 4))        ;  then next 4
           (= (substr code 6 1) "-")))      ;  and hyphen between
        (t nil)))                           ;Otherwise, not valid

(defun check (code)
  (eq (type (read code)) 'int))             ;Is this a number?
```

6. `stoa` (***S***tring ***TO A***SCII) converts a string into a list of its ASCII values.

```
(defun stoa (str / cnt l)
  (setq cnt 1)
  (while (<= cnt (strlen str))                    ;While more characters
    (setq l (cons (ascii (substr str cnt 1)) l)   ;Get next, convert it, cons onto list
          cnt (1+ cnt)))                          ;Inc to get next char
  (reverse l))
```

7. `c:nl-seq` prompts the user for a starting number or letter. On each pass through a loop, it prompts for a point and places the next number or letter in the sequence at the supplied point.

```
(defun c:nl-seq (/ inp asc txt char-flag pt)
  (setq inp (getstring "\nEnter starting number or letter: "))
  (if (not (numberp (setq num (read inp))))       ;Did user enter number?
      (setq char-flag t                           ;No, record fact and
            asc (ascii inp)))                     ;  get ASCII value of character
  (while (setq pt (getpoint "\nEnter point: "))   ;Get next point
    (if char-flag                                 ;If user entered an alpha
        (setq txt (chr asc))                      ;Convert ASCII to letter
        (setq txt (rtos num 2 0)))                ;Else convert num to string
    (command "text" "mc" pt "" "" txt)            ;Put in drawing
    (if char-flag                                 ;If user entered an alpha
        (setq asc (1+ asc))                       ;Increment ASCII value
        (setq num (1+ num)))))                    ;Else, increment number
```

8. `surround` takes two coordinates expressed as strings and puts in the drawing a table showing the entered coordinates as well as the sectors surrounding those coordinates.

```
(defun surround (n1 n2 / p b)
  (setq n1 (1- (atoi n1))                          ;Convert strings to #
        n2 (1+ (atoi n2)))                         ; and initialize
  (setq p (getpoint "Select starting point for table: "))
  (repeat 3                                        ;Outer loop
    (terpri)
    (repeat 3                                      ;Inner loop
      (command "text" p "" ""
        (strcat (itoa n1) "-" (itoa n2) " "))      ;Print info
      (setq n1 (1+ n1))                            ;Increment n1's value to next
      (setq p (list (+ (car p) 1.5) (cadr p)))     ;Move x of input point
    )
    (setq n2 (1- n2)                               ;Decrement n2's value to previous
          p  (list (- (car p) 4.5)                 ;Reset x
                   (1- (cadr p)))                  ;Decrement y of input point
          n1 (- n1 3))                             ;Reset n1
  )
  (prin1))
```

9. `tab` takes an AutoLISP object and a count. It returns a string containing a number of spaces equal to the count minus the number of characters in the object. It calls the `otos` function we wrote in Chapter 8, Lab Exercise 5.

```
(defun tab (obj num / spaces f len diff)
  (setq spaces ""                              ;Initialize
        len    (strlen (otos obj))             ;Convert object to string and find length
        diff   (- num len))                    ;Max num of spaces minus length
  (if (not (minusp diff))                      ;If neg, object was too long anyway
      (repeat diff                             ;Make string of spaces
        (setq spaces (strcat " " spaces))))    ;Add another space to front of string
  spaces)                                      ;Return string of spaces
```

10. `columns` takes a list and a tab setting and prints the list elements on the screen in columns.

```
(defun columns (l n)
  (terpri)
  (foreach elt l                ;Get each element in turn
    (princ elt)                 ;Print it
    (princ (tab elt n)))        ;Tab over specified amount
  (princ))
```

11. `c:syms` lists all the bound symbols and what kind of data they are bound to. It finds the datatype of the value of each symbol and conses the symbol itself onto the proper list. When done, it prints out each list. (`c:syms` is based on a program by Todd Moen.)

```
(defun c:syms (/ kind ilst rlst slst stlst llst flst elst plst)
  (foreach item (atoms-family 0)                          ;Look at each symbol
    (setq kind (type (eval item)))                        ;What datatype is its value?
    (cond ((= kind 'int)      (setq ilst (cons item ilst))) ;If int, cons it on
          ((= kind 'real)     (setq rlst (cons item rlst)))
          ((= kind 'sym)      (setq slst (cons item slst)))
          ((= kind 'str)      (setq stlst (cons item stlst)))
          ((= kind 'file)     (setq flst (cons item flst)))
          ((= kind 'ename)    (setq elst (cons item elst)))
          ((= kind 'pickset) (setq plst (cons item plst)))
          ((and (= kind 'list) (numberp (car (eval item))))
                              (setq llst (cons item llst)))))
  (princ "\nIntegers: ")          (princ ilst)  ;Print symbols whose values are integers
  (princ "\nReal numbers: ")      (princ rlst)
  (princ "\nSymbols: ")           (princ slst)
  (princ "\nStrings: ")           (princ stlst)
  (princ "\nPoint Lists: ")       (princ llst)
  (princ "\nFile Descriptors: ") (princ flst)
  (princ "\nEntity Names: ")      (princ elst)
  (princ "\nSelection Sets: ")    (princ plst)
  (princ))
```

12. `c:port` makes the viewpoint whose lower left corner is at 0,0 be the current viewport.

```
(defun c:port (/ port-id)
  (foreach port (vports)                  ;Examine each list
    (and (zerop (caadr port))             ;x-coord 0?
         (zerop (cadadr port))            ;y-coord 0?
         (setvar "cvport" (car port))))   ;New current vport
  (prin1))
```

Chapter 10. Accessing and Modifying Entities

1. a) End points: `(6 3 0)`, `(9 6 4)`, `(12 5 0)`, `(10 2 0)`

   b) Two of the lines are on Layer 1, one is on Layer 0.

   c)
   ```
   (setq ed1 (entget (car (entsel))))          ;Select a circle
   (setq ed2 (entget (car (entsel))))          ;Select other circle
   (setq c1 (cdr (assoc 10 ed1)))
   (setq ed (list '(0 . "LINE") (assoc 10 ed2) (cons 11 c1)))
   (entmake ed)
   ```

   Building this list is tricky because the value of `c1`, which is the 10 field of the circle, is being used as the 11 field of the line. This means that we can't supply a quoted list because `c1` must be evaluated. If we enter `'(11 . c1)`, AutoLISP will build a dotted pair whose cdr is the symbol `c1`. Because at least one operand needs to be evaluated, we must build the list (with the `list` function) rather than supplying a quoted list.

   d)
   ```
   (entdel (entlast))
   ```

   e)
   ```
   (setq ed3 (entget (car (entsel))))          ;Select line on right
   (setq ed3 (subst '(11 6 3) (assoc 11 ed3) ed3))
   (entmod ed3)
   ```

   f) Assign to `e6` the polyline header entity name. Assign to `ed7` and `ed8` the entity data for the first two vertex subentities.

   ```
   (setq ed7 (subst '(10 3 3) (assoc 10 ed7) ed7))
   (entmod ed7)
   (setq ed8 (subst '(10 3 5) (assoc 10 ed8) ed8))
   (entmod ed8)
   (entupd e6))
   ```

2. `lastent` finds the last main or subentity in the drawing. This solution is a simplified version of the one in the `entlast` section of the *AutoLISP Reference.*

   ```
   (defun lastent (/ a)
     (setq a (entlast))                        ;Get last main entity
     (while (entnext a)                        ;If subentities follow,
       (setq a (entnext a)))                   ; loop until no more
     a)                                        ;Return last entity
   ```

3. `sub` shortcuts the data replacement process for entities.

```
(defun sub (code new ed)
  (entmod (subst (cons code new)
                 (assoc code ed)
                 ed)))
```

4. `c:nl` and `c:nlx` prompt the user for an entity and layer name, then move the selected entity to the chosen layer.

```
(defun c:nl (/ ed layer)
  (setq ed    (entget (car (entsel)))        ;Select entity and get edata
        layer (getstring "\nNew layer for this object: ")
        ed    (subst (cons 8 layer)          ;Alter data to new layer
                     (assoc 8 ed) ed))
  (entmod ed)
  (princ))

;;;This is a compacted version that uses the sub function from Exercise 3.
(defun c:nlx (/ ed)
  (setq ed (entget (car (entsel))))
  (sub 8 (getstring "\nNew layer for this object: ") ed)
  (princ))
```

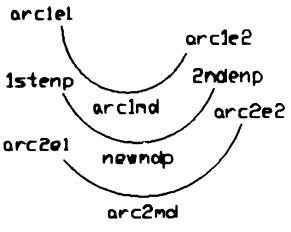
5. `c:cc` changes the color of selected entities. (`c:cc` is based on a program by Jim Porter.)

```
(defun c:cc (/ clr match layr)
  (initget "Red Yellow Green Cyan Blue Magenta White byLayer byEntity")
  (setq clr (getkword
             "\nRed/Yellow/Green/Cyan/Blue/Magenta/White/byLayer/byEntity: "))
  (prompt"\nSelect item(s) to change color of: ")
  (command "select" pause)                              ;Allow user to choose entities
  ;;On byEntity the target entity might not have a color (62) field.
  ;;Then we must learn the entity's layer (8) and search the Layer table.
  (if (= clr "byEntity")
      (progn (setq match (car (entsel
                "\nPick entity with desired color for matching: ")))
        (setq clr (cdr (assoc 62 (entget match))))      ;Get data, color field
        (if (null clr)                                  ;If not present
            (setq layr (cdr (assoc 8 (entget match)))   ;8 field of target entity
                  layr (tblsearch "layer" layr)         ;Get Layer table entry
                  clr  (cdr (assoc 2 layr))))))         ;Get Layer's color
  (command "change" "p" "" "p" "c" clr ""))             ;Change color property of ents
```

6. `c:midln2` is a version of the `c:midln` command in Chapter 9. It uses entity functions to determine whether the user selected a line (and re-prompts if she didn't) and to access the starting and ending point of each line. (`c:midln2` is based on a program by Todd Moen.)

```
(defun c:midln2 (/ ed1 ln1 ln1e1 ln1e2 ed2 ln2 ln2e1 ln2e2 temp 1stmp 2ndmp)
  ;;Reprompt if user selected wrong entity type or nothing at all
  (setq ln1 (entsel "\nSelect first line: "))
  (while (or (null ln1)                                    ;Add test for line selection
             (not (eq (cdr (assoc 0 (setq ed1 (entget (car ln1))))) "LINE")))
    (setq ln1 (entsel "\nEntity is not a line. Select first line: ")))

  (redraw (car ln1) 3)                                     ;Highlight first line
  (setq ln2 (entsel "\nSelect second line: "))
  (while (or (null ln2)
             (not (eq (cdr (assoc 0 (setq ed2 (entget (car ln2))))) "LINE")))
    (setq ln2 (entsel "\nEntity is not a line. Select second line: ")))

  (setq ln1e1 (cdr (assoc 10 ed1))                         ;Get both endpoints
        ln1e2 (cdr (assoc 11 ed1))                         ;  for each line
        ln2e1 (cdr (assoc 10 ed2))                         ;  using entity functions
        ln2e2 (cdr (assoc 11 ed2)))

;;;From this point on the routine is the same as c:midln
 ;;make ln2e1 be the ln2 endpt nearest to the ln1 pick point.
  (if (> (* (distance ln1e1 ln2e1)                         ;Is product of opposite
            (distance ln1e2 ln2e2))                        ;  distances greater?
         (* (distance ln1e1 ln2e2)                         ;(cross product of vectors)
            (distance ln1e2 ln2e1)))
      (setq temp  ln2e1                                    ;Yes, switch ln2e1 and ln2e2
            ln2e1 ln2e2
            ln2e2 temp)
  )
  (setq 1stmp (midpoint ln1e1 ln2e1)                       ;Set endpoints of midline
        2ndmp (midpoint ln1e2 ln2e2))
  (command "line" 1stmp 2ndmp "")                          ;Draw midline
  (redraw (car ln1))                                       ;Unhighlight line
  (prin1))
```

7. `c:midarc` draws an arc midway between two other arcs. The following picture shows the settings of the variables used in the command. (`c:midarc` is based on a program by Todd Moen.)



```
(defun c:midarc (/ ar1 sa1 ea1 rd1 rp1 arc1e1 arc1e2 arc1md 1stenp
                   ar2 sa2 ea2 rd2 rp2 arc2e1 arc2e2 arc2md 2ndenp newmdp)
  ;;Reprompt if user selected wrong entity type or nothing at all
  (setq ar1 (entsel "\nSelect first arc: "))
  (while (or (null ar1)
             (not (eq (cdr (assoc 0 (setq ent1 (entget (car ar1))))) "ARC")))
    (setq ar1 (entsel "\nEntity is not an arc. Select first arc: ")))
  (redraw (car ar1) 3)                                  ;Highlight first arc
  (setq sa1    (cdr (assoc 50 ent1))                    ;Starting angle
        ea1    (cdr (assoc 51 ent1))                    ;Ending angle
        rd1    (cdr (assoc 40 ent1))                    ;Radius of arc
        rp1    (cdr (assoc 10 ent1))                    ;Center point of arc
        arc1e1 (polar rp1 sa1 rd1)                      ;First ending point
        arc1e2 (polar rp1 ea1 rd1)                      ;Second ending point
        arc1md (osnap (cadr ar1) "mid"))                ;Midpoint of arc

  (setq ar2 (entsel "\nSelect second arc: "))
  (while (or (null ar2)
             (not (eq (cdr (assoc 0 (setq ent2 (entget (car ar2))))) "ARC")))
  (setq ar2 (entsel "\nEntity is not an arc. Select Second Arc: ")))
  (setq sa2    (cdr (assoc 50 ent2))                    ;Starting angle
        ea2    (cdr (assoc 51 ent2))                    ;Ending angle
        rd2    (cdr (assoc 40 ent2))                    ;Radius of arc
        rp2    (cdr (assoc 10 ent2))                    ;Center point of arc
        arc2e1 (polar rp2 sa2 rd2)                      ;First ending point
        arc2e2 (polar rp2 ea2 rd2)                      ;Second ending point
        arc2md (osnap (cadr ar2) "mid"))                ;Midpoint of arc

  (setq 1stenp (midpoint arc1e1 arc2e1)                 ;First endpoint of new arc
        2ndenp (midpoint arc1e2 arc2e2)                 ;Second endpoint of new arc
        newmdp (midpoint arc1md arc2md))                ;Midpoint of new arc
  (command "ARC" 1stenp newmdp 2ndenp)                  ;Draw new arc
  (redraw (car ar1))                                    ;Unhighlight first arc
  (prin1))
```

8. `join-pts` creates a closed polyline connecting all the points in the supplied list.

```
(defun join-pts (pts)
  (entmake '((0 . "polyline") (70 . 1)))                ;Create header
  (foreach pt pts                                       ;Create all vertices
    (entmake (list '(0 . "vertex") (cons 10 pt))))
  (entmake '((0 . "seqend")))                           ;Tack on seqend
  (princ))
```

9. `c:ents` labels the database of existing entity names `e1, e2, ... en`, and their associated data `ed1, ed2, ... edn`. Note that `e1, ed1`, and all other newly created variables are global.

```
  (defun c:ents (/ n ex ey edx)
1   (setq ey  (entnext)
          e1  ey                                    ;Init first entity name
          ed1 (entget e1)                           ;Init first entity data
          n   2)                                    ;Start calculating from 2nd
2   (while (entnext ey)                             ;While next entity non-nil,
      (setq ex (read (strcat "e" (itoa n))))        ;"e2" -> e2
      (set ex (entnext ey))                         ;Ename of e2
3     (setq edx (read (strcat "ed" (itoa n))))      ;"ed2" -> ed2
      (set edx (entget (eval ex)))                  ;Edata for e2
      (setq ey (eval ex)
            n  (1+ n))                              ;Inc index
     ex))                                           ;Return last ename
```

1. The `c:ents` command initializes values for `e1` and `ed1` and begins the loop with `n = 2`.

2. On its first pass the `while` loop makes the value of `e2` be the entity name of the second entity. `itoa` converts 2 to `"2"`, `strcat` concatenates `"e"` onto the front, `read` converts `"e2"` to a symbol, and `set` binds the symbol to the entity name. Since the symbol name is being built by the code, `set` must be used in place of `setq`.

3. It follows similar steps to build the symbol `ed2` and assign it the second entity data as a value. It then gets the next entity name, increments the counter, and repeats the loop until all entities have been assigned to symbols. The last entity name is returned so the user learns how many entities are in the drawing.

10. `handout` accesses each entity in the supplied selection set and conses its handle onto a list. It then writes the completed list to a file. It uses the name of the drawing as the default filename, which associates the data file with its related drawing.

```
(defun handout (ss / cnt e ed eh l f)
  (setq cnt (sslength ss))                          ;How many entities in SS?
  (while (> cnt 0)                                  ;Sequence through SS
    (setq e  (ssname ss (setq cnt (1- cnt)))        ;Access each entity in turn
          ed (entget e)                             ;Fetch entity data
          eh (cdr (assoc 5 ed))                     ;Get handle
          l  (cons eh l)))                          ;Cons handle onto list
  (setq f (open (getfiled "FILE TO WRITE" (getvar "dwgname") "dat" 1) "w"))
  (prin1 l f)                                       ;Write complete list to file
  (setq f (close f)))
```

`handin` reads a list of handles from a file and returns a selection set containing the "handled" entities. Remember to run this function inside of a `setq` to save the returned selection set.

```
(defun handin (/ ss f l)
  (setq ss (ssadd)                                    ;Create empty SS
        f  (open (getfiled "FILE TO READ" (getvar "dwgname") "dat" 4) "r")
        l  (read (read-line f))                       ;Read list of handles
        f  (close f))
  (foreach eh l                                       ;Access each handle in turn
    (ssadd (handent eh) ss))                          ;Add entity name to SS
  ss)                                                 ;Return SS
```

`handin2` is an alternative solution using `mapcar`. Since `mapcar` returns a list containing the result, we must take the car to get the selection set.

```
(defun handin2 (/ ss f l)
  (setq ss (ssadd)                                    ;Create empty SS
        f  (open (getfiled "FILE TO READ" (getvar "dwgname") "dat" 4) "r")
        l  (read (read-line f))                       ;Read list of handles
        f  (close f))
  (car (mapcar '(lambda (eh)
                  (ssadd (handent eh) ss))
               l)))
```

11. `c:xdalt` asks the user to select an entity and input an application ID, then adds the supplied xdata to the existing entity data.

```
(defun c:xdalt (/ e id ed l1 l2 l3 typ data)
  (setq e  (car (entsel "\nSelect entity to modify: "))
        id (getstring "Application ID: ")
        ed (entget e (list id)))                      ;Get data with Xdata
  (regapp id)                                         ;In case ID doesn't exist yet
  (initget "String Real")
  (setq typ (getkword "Type of data to add (<String> Real): "))
  (if (eq typ "Real")
      (setq data (cons 1040 (getreal "Enter real number: ")))
      (setq data (cons 1000 (getstring t "Enter string: "))))
  ;;Build new Xdata. l1 is previous data, l2 is new and old, l3 adds on -3 and AppID.
  (setq l1 (cdadr (assoc -3 ed))                      ;((1040 . 55.6)) - Previous data
        l2 (append l1 (list data))                    ;((1040 . 55.6) (1000 . "abcde"))
        l3 (list -3 (cons id l2)))                    ;(-3 ("rh-test" (1040 . 55.6) . . . ))
  (entmod (append (entget e) (list l3)))
  (princ))
```

Notes

- This command accesses the entity name (unlike `c:xdadd`, which asks for the entity data) because it must call `entget` to get the extended entity data.
- The original intent of this function was to modify existing Xdata for a particular application ID. Upon testing, it was discovered that the command works even if the entity doesn't have Xdata with the specified application ID. In that case `l1` is set to `nil`, `nil` is appended to the new list, and `l2` is bound to the new list alone, without an original Xdata list. In short, Xdata is created anew rather than appended to existing Xdata.

  As it turns out, this function encompasses the task of the `c:xdadd` command and renders it obsolete. Sometimes our most effective results weren't our original intentions, so it pays to test functions thoroughly.

In Section 12.4 `c:xdalt` is used to illustrate a very tricky error. You might consider studying the error while this command is fresh in your mind.

12. `c:unblk` unblocks the selected block and adds identifying Xdata to each new entity. (`c:unblk` and `c:reblk` are based on a concept by Todd Moen.)

```
  (defun c:unblk (/ las data app e ed)
    (setq las    (entlast)                                ;EOF marker
1         #xm    (strcat "rh-" (substr (rtos (getvar "cdate") 2 6) 12))
                        ;Unique xdata marker: "rh-1734"; number is current seconds
3         data   (cons 1000 "a")                          ;Create dummy dotted pair
          app    (list -3 (list #xm data)))
    (while (null (regapp #xm))                            ;Appid already used, get another
      (setq #xm (strcat "rh-" (substr (rtos (getvar "cdate") 2 6) 12))))
    (setq e   (car (entsel "Select block to explode: "))
          ed  (entget e)
2         #nm (cdr (assoc 2 ed))                          ;Name of block
          #pt (cdr (assoc 10 ed)))                        ;Block insertion point
4   (command "explode" e)
    (while (setq las (entnext las))                       ;Sequence through exploded ents
      (setq ed (entget las))                              ;Get their data
      (entmod (append ed (list app))))                    ;Add xdata to them
    (princ))
```

Notes

1. A primary consideration when working with Xdata is choosing a unique application ID. Otherwise we might subsequently access the wrong entities. This line reads the current date and time and converts it to a string.

Unfortunately, it can't use this string as is because it contains a period. Since periods are illegal in application ID strings (despite the example to the contrary shown with the `regapp` function description in the *AutoLISP Reference*), this code must use a substring.

The substring comprises four digits representing seconds, because these change frequently and are likely to be unique. This smaller string is concatenated to the author's initials to complete the application ID.

For total security, when the application ID is registered a few lines hence, the code checks whether `regapp` returns `nil`. If it does, the application ID has already been registered, so a new one is built. This is probably overkill, but it's good to see various approaches to building a unique application ID.

2. The application ID, block name, and insertion point are saved in the global variables `#xm`, `#nm`, and `#pt`, respectively. The unusual names were chosen to prevent them from being accidentally rebound. If the user runs `c:unblk` on a second insert, he must first save the values of these variables elsewhere because they will be overwritten.

3. When we build an extended entity data record, we're required to supply at least one Xdata dotted pair. This line sets the variable `data` to this dummy pair, but the routine never looks at it because it's only interested in the application ID.

4. The EXPLODE command behaves differently when executed within AutoLISP than when we call it from the Command: prompt. When called within a `command` function it only accepts a single block. Therefore, a null string at the end of the call, for example `(command "explode" e "")`, is unnecessary. In fact, if the null string is present, `command` will probably issue an "Unknown command" error, but the program will run correctly.

`c:reblk` reblocks the entities whose Xdata Application ID matches the value of the variable `#xm`.

```
(defun c:reblk (/ ss)
  (setq ss (ssget "x" (list (list -3 (list #xm)))))
  (if ss                                        ;Xdata found
      (command "block" #nm "y" #pt ss ""        ;Block selected entities, and insert
               "insert" #nm #pt "" "" ""))      ;  block using name saved from c:unblk
  (princ))
```

`c:reblk2` is another algorithm for accomplishing the same task. An alternative to using `ssget` is to build the selection set ourselves.

```
(defun c:reblk2 (/ ss e ed xm)
  (setq ss (ssadd)                          ;Create empty ss
1       e  (entnext)                        ;Get first ename
        ed (entget e (list #xm))            ;Get first edata
        xm (assoc -3 ed))                   ;Check for xdata
2 (while (null xm)                          ;Loop until xdata found
    (if (null (setq e (entnext e)))         ;Get next ename. If at end
        (setq xm t)                         ;  set up to break out of loop
        (progn                              ;  Else
          (setq ed (entget e (list #xm))    ;Get next edata
                xm (assoc -3 ed)))))        ;Check that for xdata
  (if (eq xm t)                                 ;Xdata not found; no ents
      (princ "\nNo entities found with this -3 code")
      (progn                                    ;Else found xdata
3       (while xm                               ;Foreach ent with this xdata
          (ssadd e ss)                          ;Add ename to ss
          (setq e (entnext e))                  ;Check next ename
          (if (null e)                          ;If last entity
              (setq xm nil)                     ;We'll quit the loop
              (setq ed (entget e (list #xm))    ;Else, get its edata
                    xm (assoc -3 ed))))         ;  and its xdata
4       (command "block" #nm "y" #pt ss "")     ;Block selected entities
        (command "insert" #nm #pt "" "" "")))   ;  and insert block. Use name and
  (princ))                                      ;  insert pt saved from c:unblk
```

1. The initial settings of the variables `e`, `ed`, and `xm` are probably unnecessary. The only way the first entity could be a search item is if our drawing contained only the one exploded block and nothing else . . . a highly unlikely occurrence. We're just being thorough here.

2. The first `while` loop searches sequentially through the entity database until it finds an entity with the proper application ID. It may not find such an entity, so it must also check if it's at the end of the database and, if so, exit the loop. Otherwise the loop will never end.

3. Once the first sought-after entity is found, the others will immediately follow. The second `while` loop sequences through subsequent entities, adding each to the selection set, until one is found without the proper application ID.

4. After the selection set is built it's handed to the BLOCK command. The block is re-created and inserted into the drawing.

Chapter 11. Selection Sets and Nongraphical Entities

1. a)
```
Command: (setq ss1 (ssget "x"))
Command: (sslength ss1)
19
```

b)
```
Command: (setq e1 (entnext) e2 (entnext e1))
Command: (ssdel e1 ss1)
Command: (ssdel e2 ss1)
```

c)
```
Command: (setq ss2 (ssadd e1))
Command: (ssadd e2 ss2)
```

d)
```
Command: (setq ss3 (ssget "x" '((0 . "circle"))))
Command: (sslength ss3)
8
```

e)
```
Command: (setq ss4 (ssget "p" '((8 . "1"))))
Command: (command "move" ss4 "" '(0 0) '(0.5 0))
```

f)
```
Command: (setq ss5 (ssget "x" '((0 . "text") (-4 . "<") (40 . 0.4))))
```

Note that we can't just say

```
Command: (setq ss5 (ssget "x" '((-4 . "<") (40 . 0.4))))
```

because circles' radii and polylines' starting width are also represented by a group code of 40.

g)
```
(defun c:set6 ()
  (setq ss6 (ssget "x"
    '((-4 . "<or")
        (-4 . "<and")
          (0 . "circle")              ;Circles
          (-4 . ">")                  ;  with radius greater than 0.5
          (40 . 0.5)
        (-4 . "and>")
        (-4 . "<and")
          (0 . "text")
          (1 . "This is Text")
        (-4 . "and>")
      (-4 . "or>")))))
```

h) **Command:** `(setq ss7 (ssget "x" '((1 . "This is Text"))))`

or

Command: `(setq ss7 (ssget "p" '((0 . "text"))))`

2. `c:f-c` adds all the circles in the drawing to a selection set named by the user. It also prints the center point, radius, and layer of each. The changes from `c:find-circles` are in uppercase.

```
(defun c:f-c (/ en ed ss obj)
  (setq en (entnext)
        SS (READ (GETSTRING "\SELECTION SET NAME: ")))
  (SET SS (SSADD))                                    ;CREATE EMPTY SELECTION SET
  (terpri)
  (princ "     Center point              Radius         Layer")   ;ADD LAYER TO HEADING
  (terpri)
  (while en                                           ;en not nil--it's bound to ename
    (setq ed (entget en))
    (if (eq (cdr (assoc 0 ed)) "CIRCLE")              ;type = circle?
        (progn
          (SSADD EN (EVAL SS))                        ;ADD ENAME TO SS
          (setq obj (print (cdr (assoc 10 ed))))      ;Center
          (princ (tab obj 30))                        ;Tab over
          (setq obj (prin1 (cdr (assoc 40 ed))))      ;Radius
          (PRINC (TAB OBJ 15))                        ;TAB AGAIN
          (PRIN1 (CDR (ASSOC 8 ED)))))                ;LAYER NAME
    (setq en (entnext en)))                           ;Get next until nil
  (prin1))
```

3. `c:cl` moves the selected entities to the current layer.

```
(defun c:cl (/ ss)
  (setq ss (ssget))
  (command "change" ss "" "p" "la" (getvar "clayer") "")
  (prin1))
```

4. `ss-sieve` takes a selection set and an entity type as arguments and removes all entities that are of that type. (`ss-sieve` was written by Bill Kramer.)

```
(defun ss-sieve (ss item / count ent)
  (setq count (sslength ss))                          ;Number of members
  (while (> count 0)
    (setq ent (entget (ssname ss                      ;Get entity indexed
                      (setq count (1- count)))))      ;  by (1- count)
    (if (eq (cdr (assoc 0 ent))                       ;Entity type same as
            (strcase item))                           ;  search item?
        (ssdel (ssname ss count) ss))))               ;Yes, delete it
```

This algorithm uses ssname in an interesting manner. It sets count equal to the number of entities in the selection set and uses that number as ssname's index. As count is decremented on each pass through the loop, ssname accesses the selection set from back to front. Since the order of entity names in a selection set doesn't matter, this is a perfectly valid approach.

Following is an alternative solution. It uses the SELECT command to build a selection set of all entities in the supplied selection set solely to enable use of the ssget "P" option. Since it builds a new selection set, it must be run inside of a setq.

```
(defun sieve (ss item)
  (command "select" ss "")                    ;To have "Previous" to give to ssget
  (ssget "p" (list '(-4 . "<not") (cons 0 item) '(-4 . "not>"))))
```

sieve2 is the same as sieve except that it builds a selection set of entities that *are* of the specified type.

```
(defun sieve2 (ss item)
  (command "select" ss "")                    ;To have "Previous" to give to ssget
  (ssget "p" (list (cons 0 item))))           ;Build ss of specified item
```

5. c:ped alters polyline widths. This version will work in both Release 12 and Release 13. (c:ped is based on a program by David Hill.)

```
(defun c:ped (/ ss thk c ed)
  (setq ss  (ssget '((0 . "polyline")))                          ;Select plines to change
        thk (getdist "Enter new polyline width: ")
        c   0)                                                   ;Counter
  (while (< c (sslength ss))                                     ;Change selected plines
    (setq e (ssname ss c))                                       ;Get ename
    (while (/= (cdr (assoc 0 (setq ed (entget e)))) "SEQEND")    ;  & edata until seqend
      (setq ed (subst (cons 40 thk) (assoc 40 ed) ed)            ;Alter vertex'
            ed (subst (cons 41 thk) (assoc 41 ed) ed))           ;  40 and 41 fields
      (entmod ed)                                                ;Update database
      (entupd e)                                                 ;Redraw pline
      (setq e (entnext e)))
    (setq c (1+ c)))                                             ;Get next entity
  (princ))
```

6. `c:blkscale` determines whether the user wants to select individual blocks to scale or scale all inserts of a selected block.

   If the former, it supplies `scl-blk` with block inserts that the user selects with `ssget`. If the latter, it supplies `scl-blk` with all inserts for a block the user selects. (`c:blkscale` is based on a program by Tony Palmisano.)

```
(defun c:blkscale (/ sblk bnam)
  (if (y-np "\nChange all occurrences of the Block?")
      (progn                                          ;Yes
        (setq sblk (entget (car (entsel "\nSelect one of the Blocks: ")))
              bnam (cdr (assoc 2 sblk)))              ;Get block name
        (scl-blk (ssget "x" (list (cons 2 bnam)))))   ;Make SS of all inserts
      (scl-blk (ssget '((0 . "insert"))))             ;No, SS of chosen inserts
  (princ))
```

   This pretty much does the same task as the `c:sclblk` command in Section 11.2.1:

```
(defun scl-blk (ss / sf inc blk insp)
  (initget 7)
  (setq sf  (getreal "\nScale Factor (1 equals no change): ")
        inc -1)
  (repeat (sslength ss)                              ;Get each block in SS
    (setq blk  (ssname ss (setq inc (1+ inc)))       ;Block's ename
          insp (cdr (assoc 10 (entget blk))))        ;Block's insert point
    (command "scale" blk "" insp sf)))               ;Scale the block
```

7. `c:lay-num` returns a count of the number of layers in a drawing.

```
(defun c:lay-num (/ count)
  (setq count 1)                        ;Initialize count
  (tblnext "layer" t)                   ;Reset table and get Layer 0 info
  (while (tblnext "layer")              ;While more layers,
    (setq count (1+ count)))            ;  increment count
  count)                                ;Return count
```

8. `tlist` takes the name of a symbol table and prints all of the entries in that table.

```
(defun tlist (table / entry)
  (textscr)                             ;Show text screen
  (print (tblnext table t))             ;Print first entry in the table
  (while (setq entry (tblnext table))   ;Get next entry until nil
    (print entry))                      ;  and print it.
  (princ))
```

9. `c:col` returns the color of the selected entity, even if it is BYLAYER.

```
(defun c:col (/ ed c l)
  (setq ed (entget (car (entsel))))                    ;Get edata for selected entity
  (if (setq c (cdr (assoc 62 ed)))                     ;If edata contains color,
      c                                                ;  return color. Else,
      (progn (setq l (cdr (assoc 8 ed)))               ;Get layer and
             (cdr (assoc 62 (tblsearch "layer" l))))))  ;Return color from layer table
```

10. `c:nl2` prompts the user for an entity and layer name, then moves the selected entity to the chosen layer. It calls `get-prompt` to make a string of existing layer names for the prompt.

```
(defun c:nl2 (/ ed layer prompt)
  (setq ed     (entget (car (entsel)))          ;Select entity, get edata
        prompt (get-prompt))                    ;Obtain string of layer names
  (initget prompt)                              ;Limit user to these
  (setq layer (getkword                         ;Prompt user with choices
                (strcat "\nNew layer for this entity <" prompt ">: "))
        ed    (subst (cons 8 layer)             ;Alter data to new layer
                     (assoc 8 ed) ed))
  (entmod ed)                                   ;Update database
  (princ))
```

`get-prompt` cycles through the Layer table and builds a prompt string consisting of the existing layer names separated by spaces.

```
(defun get-prompt (/ prompt layer)
  (setq layer  (tblnext "layer" t)              ;tblnext t rewinds table
        prompt (cdr (assoc 2 layer)))           ;Get first layer name
  (while (setq layer (tblnext "layer"))         ;Another entry in table
    (setq prompt (strcat prompt " "             ;Append prompt and " "
                   (cdr (assoc 2 layer)))))     ;  to next layer name
  prompt)                                       ;Return prompt
```

11. `c:fl` makes a list of the names of all frozen layers. It uses the global variable `#l-list` to save the list for later.

```
(defun c:fl (/ tab flag)
  (setq #l-list nil)                            ;Reset #l-list
  (setq tab (tblnext "layer" t))                ;Get first table entry
  (while tab
    (setq flag (cdr (assoc 70 tab)))            ;Get 70 field of entry
    (if (= (logand flag 1) 1)                   ;If bit 0 = 1, add layer name to list
        (setq #l-list (cons (cdr (assoc 2 tab)) #l-list)))
    (setq tab (tblnext "layer")))               ;Get next table entry
  #l-list)                                      ;Return list of frozen layers
```

`c:rl` returns layers to the status remembered in `#l-list`. This command can't be run from a layer that will become frozen!

```
(defun c:rl ()
  (command "layer" "t" "*" "")                    ;Thaw all layers
  (foreach l #l-list                              ;Freeze layers remembered in #l-list
    (command "layer" "f" l ""))
  #l-list)                                        ;Return list of frozen layers
```

12. `c:ns` uses `entmake` to create a new text style based on STANDARD.

```
(defun c:ns (/ e ed name height width)
  (setq e     (tblobjname "style" "standard")          ;Standard style ename
        ed    (entget e)                               ;Get its data
        name  (getstring "\nNew style name: "))        ;Get new style name
  (setq ed (subst (cons 2 name) (assoc 2 ed) ed))      ;Change name
  (initget 7)
  (setq height (getdist "\nHeight: "))
  (setq ed (subst (cons 40 height) (assoc 40 ed) ed))  ;Change height
  (initget 7)
  (setq width (getdist "\nWidth: "))
  (setq ed (subst (cons 41 width) (assoc 41 ed) ed))   ;Change width
  (entmake ed)                                         ;Create new style
  (setvar "textstyle" name)                            ;Make it current
  (princ))
```

`c:ns2` does the same as `c:ns` but creates entity data from scratch.

```
(defun c:ns2 (/ ed name height width)
  (setq name  (getstring "\nNew style name: "))        ;Get new style name
  (initget 7)
  (setq height (getdist "\nHeight: "))                 ;Height
  (initget 7)
  (setq width (getdist "\nWidth: "))                   ;Width
  ;;Give entmake data necessary to create a style entry.
  (entmake (list '(0 . "style") '(100 . "AcDbSymbolTableRecord")
                 '(100 . "AcDbTextStyleTableRecord") (cons 2 name) '(70 . 64)
                 (cons 40 height) (cons 41 width) '(3 . "txt")))
  (setvar "textstyle" name)                            ;Make it current
  (princ))
```

Chapter 14. Computer and Programming Basics

1. Following are corrections to false statements:

   b) AutoLISP programs are stored as *ASCII* files.

   c) "Here's a good example of a *stack.*"

   d) They repeat same actions on *different* data.

   f) Code can be entered at the Command: prompt.

   j) Some computers have different byte sizes.

   k) There are *1024* bytes in a kilobyte.

   l) Control *always* flows in the direction the arrow points.

2. Fill in the blanks:

   a) The "brain" of the computer is called the *CPU.* Computers that have more than one of these can perform *multiprocessing.*

   b) Some computers give the appearance of running several programs at one time. This is called *multiprogramming* or *timesharing.*

   c) Information on a hard disk can be accessed in any order that we choose. Thus the disk is called a *random access* device. Magnetic tape, on the other hand, accesses its information in the order in which it was written. It's called a *sequential access* device.

   d) Examples of output devices are *printers, plotters, terminal screens.* Examples of input devices are *keyboards, mice, cursors.* Examples of auxiliary storage devices are *hard disks, floppies, magnetic tapes, CD-ROMs.*

   e) Assume we store, in order, the numbers 1, 2, and 3 on a stack. The first number popped will be *3.* If these same numbers are placed on a queue, then the first number retrieved will be *1.*

   f) We write functions with an editor rather than simply entering the code at the Command: prompt because (1) anything entered at the Command: prompt is lost when we exit our drawing, (2) if we make a mistake at the Command: prompt we have to retype all of the code, and (3) editors help us write correct code in the first place.

g) The process of finding and correcting errors in our code is called *debugging.*

h) A program that changes programs we write into a file containing 0s and 1s is called a *compiler.*

i) When memory fills up, the least recently accessed information is sent to *virtual memory.*

j) A *recursive* function calls itself.

Appendix

B

ASCII Codes

Dec	*Oct*	*Hex*	*Char*	*Meaning*	*Dec*	*Oct*	*Hex*	*Char*
0	000	00	NUL	Null	32	040	20	Space
1	001	01	SOH	Start of Heading	33	041	21	!
2	002	02	STX	Start of Text	34	042	22	"
3	003	03	ETX	End of Text	35	043	23	#
4	004	04	EOT	End of Transmission	36	044	24	$
5	005	05	ENQ	Enquiry	37	045	25	%
6	006	06	ACK	Acknowledge	38	046	26	&
7	007	07	BEL	Bell	39	047	27	'
8	010	08	BS	Backspace	40	050	28	(
9	011	09	HT	Horizontal Tab	41	051	29	)
10	012	0A	LF	Line Feed	42	052	2A	*
11	013	0B	VT	Vertical Tab	43	053	2B	+
12	014	0C	FF	Form Feed	44	054	2C	,
13	015	0D	CR	Carriage Return	45	055	2D	–
14	016	0E	SO	Shift Out	46	056	2E	.
15	017	0F	SI	Shift In	47	057	2F	/
16	020	10	DLE	Data Link Escape	48	060	30	0
17	021	11	DC1	Device Ctrl 1	49	061	31	1
18	022	12	DC2	Device Control 2	50	062	32	2
19	023	13	DC3	Device Control 3	51	063	33	3
20	024	14	DC4	Device Ctrl 4	52	064	34	4
21	025	15	NAK	Neg. Acknowledge	53	065	35	5
22	026	16	SYN	Synchronous Idle	54	066	36	6
23	027	17	ETB	End Tran. Block	55	067	37	7
24	030	18	CAN	Cancel	56	070	38	8
25	031	19	EM	End of Medium	57	071	39	9
26	032	1A	SUB	Substitute	58	072	3A	:
27	033	1B	ESC	Escape	59	073	3B	;
28	034	1C	FS	File Separator	60	074	3C	<
29	035	1D	GS	Group Separator	61	075	3D	=
30	036	1E	RS	Record Separator	62	076	3E	>
31	037	1F	US	Unit Separator	63	077	3F	?

Dec	*Oct*	*Hex*	*Char*	*Meaning*	*Dec*	*Oct*	*Hex*	*Char*
64	100	40	@		96	140	60	'
65	101	41	A		97	141	61	a
66	102	42	B		98	142	62	b
67	103	43	C		99	143	63	c
68	104	44	D		100	144	64	d
69	105	45	E		101	145	65	e
70	106	46	F		102	146	66	f
71	107	47	G		103	147	67	g
72	110	48	H		104	150	68	h
73	111	49	I		105	151	69	i
74	112	4A	J		106	152	6A	j
75	113	4B	K		107	153	6B	k
76	114	4C	L		108	154	6C	l
77	115	4D	M		109	155	6D	m
78	116	4E	N		110	156	6E	n
79	117	4F	0		111	157	6F	o
80	120	50	P		112	160	70	p
81	121	51	Q		113	161	71	q
82	122	52	R		114	162	72	r
83	123	53	S		115	163	73	s
84	124	54	T		116	164	74	t
85	125	55	U		117	165	75	u
86	126	56	V		118	166	76	v
87	127	57	W		119	167	77	w
88	130	58	X		120	170	78	x
89	131	59	Y		121	171	79	y
90	132	5A	Z		122	172	7A	z
91	133	5B	[		123	173	7B	{
92	134	5C	\		124	174	7C	|
93	135	5D	]		125	175	7D	}
94	136	5E	^		126	176	7E	~
95	137	5F	_		127	177	7F	DEL

Appendix

C

Files on the Enclosed Floppy

The following files are on the floppy that comes with this book:

`readme.txt`	Additions, corrections, and updates made to the book text and examples since the book went to print. Status of errors found in AutoLISP itself. *Please read this file!*
`answers.lsp`	Answers to all lab exercises.
`examples.lsp`	All examples contained in the book. A comment before each function specifies where in the book the function is explained.
`draw.lsp`	The DRAW on-line tutorial, described in Sections 2.7 and 4.3.
`efc.lsp`	The Eval Flowchart (EFC) on-line tutorial, described in Section 4.2.
`ahed.com`, `ahed.doc`	The AHED editor, described in Section 14.3.3.
`pfe.zip`	The PFE Windows-based editor, described in the `readme.txt` file.

Appendix

D

Common Error Messages

Here is a brief explanation of the error messages that you're most likely to receive and some suggestions about how to recover from these error conditions. Remember that the *first* line of code following the message *always* pinpoints the exact place where the error occurred. See Section 12.1 for a more detailed explanation.

1>

You're missing a close parenthesis or a closing double quote at the Command: prompt. `2>` means you're missing two close parentheses.

bad argument type

The function expected a certain AutoLISP object type and you gave it the wrong kind. Verify that you used `quote` properly and check your parentheses.

bad formal argument list

Same as "bad function" except that the operator is a list.

bad function

The first element of the list that you gave to `eval` (the operator) was not a function. Check your parentheses. Verify that you didn't quote the function name. Make sure that the function name itself wasn't accidentally assigned a new value.

Can't reenter AutoLISP

You tried to supply an AutoLISP expression to a subr (such as the `gets`, `entsel`, and `ssget`) that requested input.

console break

You entered an `<esc>` while a function was running.

exceeded maximum string length

A string is longer than 132 characters. You're probably missing a closing double quote in a function you called or in one you loaded.

extra right paren

The file you loaded has an unbalanced right parenthesis . . . or maybe it's missing a left parenthesis.

Function cancelled

When the function prompted you for input, you entered an `<esc>`.

incorrect number of arguments

The subr expected a certain number of arguments and you supplied the incorrect number. Check your parentheses.

invalid dotted pair

You built a cons incorrectly. This error usually arises from supplying a decimal number without the leading 0 (e.g., .4 instead of 0.4).

LOAD failed

AutoLISP can't locate the file you attempted to load. Perhaps you misspelled it or it's in the wrong directory. Maybe you forgot to give the file a `.lsp` extension when you created it.

malformed list

The file you loaded is missing a close parenthesis.

malformed string

The file you loaded is missing a closing double quote.

null function

Same as "bad function" except that the operator evaluates to `nil`.

too many/few arguments

The number of arguments you supplied doesn't match the number of parameters in the function definition. Check your parentheses.

Appendix

E

Commonly Used Entity Group Codes

The *AutoLISP Reference* contains a complete listing of group codes for all graphical and nongraphical entities. In Part IV, I endeavor to show most of the more commonly used ones. Following is a list of those codes with a brief description of their usage.

Group Code	*Meaning*
-4	Relational test or logical operator follows.
-3	Extended entity data follows.
-2	Entity name. A polyline seqend subentity uses this to indicate the entity name of the main polyline header record.
-1	Entity name.
0	A string indicating the entity type or table name.
1	A text string.
2	A name, such as a block or layer name, attribute tag, etc.
5	A string specifying the entity's handle.
6	A string specifying the entity's linetype. Present if the linetype is not BYLAYER.
7	A string specifying the text style name.
8	A string specifying the entity's layer.

10	A point, such as the starting point of a line or text, center point of a circle, or vertex of a polyline.
11	A point, such as the ending point of a line.
39	A real number specifying the entity's thickness.
40	A real number, such as a circle's radius, text height, or the starting width of a polyline. 41 is the ending width.
50	An angle, such as the starting angle of an arc. 51 is the ending angle.
62	An integer specifying the entity's color number. Present if color is not BYLAYER.
66	Set to 1 in a complex entity's header record to indicate that subentities follow.
67	Absent or 0 if model space, 1 if paper space.
70	Integer flag bit to indicate such information as a polyline is closed or a layer is frozen.

Appendix

F

AutoLISP Function Reference

The following pages provide a handy reference to the AutoLISP functions. A few appear in more than one category. Each entry consists of the function's name (in code font), the number and type of arguments that the function takes, and the function's definition. Ellipses (. . .) indicate that the function takes an unlimited number of arguments. Braces {} indicate that the argument is optional. Some abbreviations are as follows:

ang	Angle expressed in radians
alist	Association list
body	One or more functions to be evaluated in order. It returns whatever the last form evaluated returns.
edata	Entity data
ename	Entity name
file	File descriptor
form	Argument given to `eval` for evaluation
func	Function
int	Integer
num	Number
obj	Any AutoLISP object (datatype)
prompt	String message sent to the user's screen
pt	Point
ss	Selection set
str	String
sym	Symbol

Conses and Lists

`append` list . . . list	Returns a new list consisting of all of the members of the supplied lists.
`apply` func list	Runs the function on the elements of the supplied list.
`assoc` obj alist	Returns the first association list pair that has the object as its car.
`cadr` cons	Returns the car of the cdr of a cons.
`car` cons	Returns the car of a cons.
`cddr` cons	Returns the cdr of the cdr of a cons.
`cdr` cons	Returns the cdr of a cons.
`cons` obj obj	Returns a new cons with the two arguments as its car and cdr.
`last` list	Returns the last element in a list.
`length` list	Returns the number of elements in a list.
`list` obj . . . obj	Returns a new list with each of the arguments as members.
`mapcar` func list . . . list	Applies the function to the car of each list, then to the cadr of each list, and so on. It constructs and returns a list of the results.
`member` obj list	Returns the remainder of the supplied list beginning with the first occurrence of the object, or `nil` if the object is not found.
`nth` n list	Returns the *nth* element of the supplied list, starting with position 0.
`reverse` list	Returns a new list with the elements of the supplied list in reverse order.
`subst` obj1 obj2 list	Returns a new list with all occurrences of obj2 replaced by obj1 in the original list.

Arithmetic Functions

+ num . . . num	Returns the sum of successive additions.
- num . . . num	Returns the remainder of successive subtractions from the first argument.
* num . . . num	Returns the product of successive multiplications.
/ num . . . num	Returns the quotient of successive divisions.
~ int	Returns the bitwise NOT of an integer.
`1+` num	Adds 1 to its argument.
`1-` num	Subtracts 1 from its argument.
`abs` num	Returns the absolute value of a number.
`atan` num1 [num2]	Returns the arctangent of a number in radians. If the second argument is present, `atan` returns the arctangent of `(/ num1 num2)`.
`boole` bfun int . . . int	Performs a bit-by-bit comparison of the integers according to bfun, a specified Boolean function.
`cos` num	Takes an angle expressed in radians and returns its cosine.
`exp` num	Returns the natural antilog of a number.
`expt` num num	Raises the first number to the power indicated by the second.
`fix` num	Takes a number and truncates it into an integer.
`float` num	Takes a number and converts it into a real number.
`gcd` int int	Returns the greatest common denominator of two integers.
`log` num	Returns the natural log of a number.

`logand` int . . . int	Performs a bit-by-bit logical AND on each of the integers.
`logior` int . . . int	Performs a bit-by-bit logical Inclusive OR on each of the integers.
`lsh` int1 int2	Logically shifts the bits in int1 an amount indicated by int2. If int2 is positive, the bits in int1 are shifted to the left. If negative, they're shifted right.
`max` num . . . num	Returns the largest (i.e., most positive) number.
`min` num . . . num	Returns the smallest (i.e., most negative) number.
`rem` num . . . num	Returns the remainder of successive divisions.
`sin` num	Takes an angle expressed in radians and returns its sine.
`sqrt` num	Returns the square root of a number.

Strings

`strcase` str {`t`}	Returns a copy of the string with all alphabetic characters uppercase. If a second argument of `t` is supplied, the string is converted to lowercase.
`strcat` str . . . str	Returns a new string consisting of all of the characters in the supplied strings.
`strlen` str	Returns the number of characters in a string.
`substr` str int [int]	Takes a string and a starting character position and returns a portion of the string (a substring) beginning at that position. The optional third argument specifies the number of characters to be returned.
`wcmatch` str str	Compares two strings, the second of which is a wildcard pattern, and returns `t` if the first string matches the pattern.

Predicates

= obj . . . obj	Returns `t` if all of the arguments are numerically equal.
< obj . . . obj	Returns `t` if the arguments are in strictly ascending order. There are also <=, >, and >= functions.
`atom` obj	Returns `t` if the argument is an atom (not a list or a cons).
`boundp` obj	Returns `t` if the argument is a symbol with a non-`nil` value. It returns `nil` if the argument is either a symbol whose value is `nil` or an object other than a symbol.
`eq` obj obj	Returns `t` if the two arguments are the *exact same* AutoLISP object.
`equal` obj obj [num]	Returns `t` if the two arguments appear the same (i.e., have the same printed representation). The optional third argument specifies the amount by which the first two arguments can differ and still be considered equal.
`listp` obj	Returns `t` if the argument is a list or a cons.
`minusp` num	Returns `t` if the numeric argument is less than 0.
`null` obj	Returns `t` if the argument is the symbol `nil`. Otherwise it returns `nil`.
`numberp` obj	Returns `t` if the argument is a number.
`type` obj	Returns a symbol that indicates the datatype of the argument.
`wcmatch` str str	Compares two strings, the second of which is a wildcard pattern, and returns `t` if the first string matches the pattern.
`zerop` num	Returns `t` if the numeric argument is 0.

Control Structure—Special Operators

`and` form . . . form	Evaluates its forms in order. If any form returns `nil`, `and` immediately exits and returns `nil`. If all forms return non-`nil`, `and` returns `t`.
`cond` list . . . list	Evaluates the car of each list until one returns non-`nil`. It then evaluates all other forms in that list and returns whatever the last form evaluated returns.
`defun` sym list body	Defines a user function or AutoCAD command that performs the tasks specified in its body, and binds the symbol to this function.
`eval` obj	Evaluates its argument. When called explicitly, it forces an extra evaluation.
`foreach` sym list body	Executes the body once for each member of the list with sym bound to that member.
`if` test form1 [form2]	If the test returns a non-`nil` result, form1 is evaluated. If the test returns `nil`, form2, if present, is evaluated.
`lambda` list body	Takes a list of parameters and a body and defines a function that performs the tasks specified in the body. It returns the function.
`not` obj	Returns `t` if the argument is `nil`. Otherwise it returns `nil`. Used to reverse a test.
`or` form . . . form	Evaluates its forms in order. If any form returns non-`nil`, `or` immediately exits and returns `t`. If all forms return `nil`, `or` returns `nil`.
`progn` body	Evaluates its body forms in order. Used to group forms together.
`quote` obj	Takes an AutoLISP object as an operand and returns that operand unevaluated.
`repeat` int body	Evaluates its body forms int times.
`setq` sym obj . . . sym obj	The first of each pair is a symbol that isn't evaluated. The second *is* evaluated and made the value of its partner symbol. It returns the last form evaluated.
`while` test body	Repeatedly evaluates the test and body forms until the test returns `nil`.

Symbols and Variables

`atoms-family` 0/1 [list]	Returns a list of symbols present in AutoLISP. The first argument specifies whether the returned value is a list of symbols or a list of strings.
`getvar` str	Returns the value of an AutoCAD system variable.
`set` sym obj	Makes the object be the value of the symbol. It returns the object.
`setq` sym obj . . . sym obj	The first of each pair is a symbol that isn't evaluated. The second *is* evaluated and made the value of its partner symbol. It returns the last form evaluated.
`setvar` str obj	Makes the object be the value of the AutoCAD system variable.
`type` obj	Returns a symbol that indicates the datatype of the argument.

Conversions

`angtof` str [int]	Takes a string representing an angle and converts it to that angle (in radians).
`angtos` ang [int] [int]	Takes an angle (in radians) and converts it into a string according to the optional editing mode and precision.
`ascii` str	Converts the first character in the string into its numeric ASCII equivalent.
`atof` str	Converts an ASCII string into a floating point (i.e., real) number.
`atoi` str	Converts an ASCII string into an integer.
`chr` int	Takes an ASCII character code and returns its character equivalent.
`cvunit` num/list str1 str2	Converts a number or a list of numbers from the unit of measurement specified by str1 to the unit of measurement specified by str2.
`distof` str [int]	Takes a string that represents a real number and converts it to that number.
`fix` num	Takes a number and truncates it into an integer.
`float` num	Takes a number and converts it into a real number.
`itoa` int	Takes an integer and converts it into an ASCII string.
`read` str	Takes a string and returns the AutoLISP object that the string represents.
`rtos` num [int] [int]	Takes a real number and converts it into a string according to the optional editing mode and precision.
`trans` pt from to `{t}`	Translates the point from the coordinate system indicated by the second argument to the coordinate system indicated by the third argument.

Display and Device Control

`graphscr`	Switches to the graphics screen on a single-screen system and returns `nil`.
`grclear`	Clears the current viewport and returns `nil`.
`grdraw` pt pt int [int]	Draws a vector between two points in the specified color and returns `nil`.
`grread` `{t/nil}` [int] [int]	Reads values from an input device and returns a list that indicates the device and the value that it received.
`grtext` int str [int]	Takes an integer code representing the status line or a screen menu area and writes the string in that area.
`grvecs` list [list]	Draws a series of vectors between points in the supplied list and returns `nil`.
`menucmd` str	Sets or retrieves a menu item's contents or status.
`redraw` [ename] [int]	Redraws the current viewport or supplied entity.
`tablet` 0/1 [pt] [pt] [pt] [pt]	Reads the current digitizer calibrations or uses the supplied points to calibrate the tablet.
`textpage`	Switches to the text screen, clears the screen (on DOS versions of AutoCAD), and returns `nil`.
`textscr`	Switches to the text screen on a single-screen system and returns `nil`.
`vports`	Returns a list of lists that describe the current viewport configuration.

I/O

`close` file	Takes a file descriptor, closes the specified file, and returns `nil`.
`findfile` str	Searches the AutoCAD library path for the supplied filename and returns the path to the file, or `nil` if the file is not found.
`getangle` [pt] [prompt]	Returns an angle entered by the user.
`getcorner` pt [prompt]	Displays a box prompt and returns a point entered by the user.
`getdist` [pt] [prompt]	Returns a distance entered by the user.
`getfiled` str str str int	Displays a dialog box on the screen and prompts the user to choose a filename. It returns the filename or the path to the file.
`getint` [prompt]	Returns an integer entered by the user.
`getkword` [prompt]	Returns a keyword entered by the user.
`getorient` [pt] [prompt]	Returns an angle entered by the user.
`getpoint` [pt] [prompt]	Returns a point entered by the user.
`getreal` [prompt]	Returns a number entered by the user.
`getstring` `{t}` [prompt]	Returns a string entered by the user.
`initget` [int] [str]	Accepts bit and keyword arguments that control the user's input to the subsequent `get` function. It returns `nil`.
`load` str [obj]	Takes the name of an AutoLISP file and loads that file into memory. The second argument is evaluated if the load fails.
`open` str str	Opens the file whose name is given as the first argument to perform the operation specified by the second argument. It returns a file descriptor.
`prin1` obj [file]	Outputs the object to the screen or to a file. It returns the object.
`princ` obj [file]	Outputs the object to the screen or to a file. If the object is a string, `princ` "pretty prints" it. It returns the object.
`print` obj [file]	Outputs the object, preceded by a carriage return and followed by a space, to the screen or to a file. It returns the object.
`prompt` str	Outputs a string to the screen and returns `nil`.
`read-char` [file]	Reads a character entered at the keyboard or from a file and returns the decimal ASCII code for that character.
`read-line` [file]	Returns a line of text entered at the keyboard or from a file.
`terpri`	Outputs a carriage return to the screen and returns `nil`.
`write-char` int [file]	Takes the decimal ASCII code for a character and outputs the character to the screen or to a file. It returns the ASCII code.
`write-line` str [file]	Outputs a string to the screen or to a file.

Entities and Selection Sets

`dictnext` ename `{t}`	Takes the entity name of a dictionary and returns the next entry in that dictionary.
`dictsearch` ename sym `{t}`	Takes the entity name of a dictionary and the name of a dictionary entry and returns that entry.
`entdel` ename	Takes an entity name and deletes the entity from the drawing database.
`entget` ename [list]	Takes an entity name and returns the entity data. If the optional list of application names is supplied, it gets the extended entity data too.
`entlast`	Returns the name of the last nondeleted main entity in the database.
`entmake` edata	Takes an entity data list and creates the appropriate entity.
`entmod` edata	Takes entity data that has been changed and modifies the entity database to reflect that change.
`entnext` [ename]	Returns the name of the first entity in the database. If given an entity name, it returns the name of the next entity in the database.
`entsel` [prompt]	Prompts the user to select an entity and returns a list consisting of the main entity name and the pickpoint.
`entupd` ename	Takes the entity name of an entity or subentity and redraws the entire entity.
`handent` handle	Takes an entity's handle and returns its entity name.
`namedobjdict`	Returns the entity name of the current drawing's named object dictionary.
`nentsel` [str]	Prompts the user to select an entity and returns a list consisting of the entity name (main or subentity) and the pickpoint.
`nentselp` [str] [pt]	Prompts the user to select an entity and returns a list consisting of the entity name (main or subentity) and the pickpoint. If the optional point argument is supplied, `nentselp` returns a list as though the user had selected the point.
`regapp` str	Registers the supplied extended entity application ID.
`snvalid` str	Returns `t` if the argument is a valid symbol table name and `nil` otherwise.
`ssadd` [ename] [ss]	Creates a new selection set or adds an entity name to an existing selection set.
`ssdel` ename ss	Deletes the entity name from the selection set.
`ssget` [str] [pts] [alist]	Builds and returns a selection set in a manner specified by the initial mode string argument or by the standard AutoCAD Select objects: mechanism.
`sslength` ss	Returns the number of entity names in the selection set.
`ssmemb` ename ss	Returns the entity name if it is a member of the selection set and `nil` if it isn't.
`ssname` ss n	Returns the *nth* entity name in the selection set, starting with position 0.
`tblnext` tblname `{t}`	Takes a table name and returns the next entry in that table.
`tblobjname` tblname str	Takes a table name and an entry and returns the entity name for that entry.
`tblsearch` tblname `str {t}`	Takes a table name and an entry and returns that entry.
`xdroom` ename	Returns the amount of extended entity space that is available for the entity.
`xdsize` list	Takes an Xdata application list and returns the number of bytes that it will consume when appended to the entity data.

Measurements

`angle` pt pt	Returns the angle between two 2D points (in radians).
`distance` pt pt	Returns the distance between two points.
`inters` pt pt pt pt`{nil}`	Takes the endpoints of two lines and returns the point where they intersect.
`osnap` pt str	Applies an object snap mode specified in the string to the supplied point and returns the resultant point.
`polar` pt ang num	Takes a point, an angle (in radians), and a distance as arguments. It returns a point at the specified angle and distance from the supplied point.
`textbox` edata	Takes a user-defined text entity data list and returns a list containing the diagonal coordinates of a box that would enclose that text.

Query and Command Functions

`command` str obj . . . obj	Runs the supplied AutoCAD command and returns `nil`. Further arguments are the data that the command needs.
`getcfg` str	Takes the name of a variable in the `acad.cfg` file and returns its value.
`getenv` str	Takes the name of a system environment variable and returns its value.
`help` {str} {str} {str}	Takes the names of a help file and a user-defined AutoCAD command and calls up help for that command. It returns the name of the help file.
`setcfg` str str	Takes strings specifying the name of a variable in the `acad.cfg` file and a value, and assigns the value to the variable. It returns the value.
`setfunhelp` str {str} {str} {str}	Associates an AutoCAD command with a help file. It returns the command name if it succeeds, `nil` if it fails.
`ver`	Returns a string containing the current AutoLISP version number.

Error Handling

`*error*` str	User-modifiable subr that is called automatically by AutoLISP when an error occurs. It reports the error on the user's screen.
`alert` str	Displays the string in an alert box on the screen and returns `nil`.
`exit`	Aborts the current function and returns the error message "quit / exit abort".
`quit`	Aborts the current function and returns the error message "quit / exit abort".
`trace` func . . . func	Traces entry to and exit from the supplied functions.
`untrace` func . . . func	Turns off tracing for the specified functions.

Application Handling

`ads`	Returns a list containing the names of all loaded ADS applications.
`autoload` str list	Takes a filename and a list of user-defined AutoCAD commands and loads the file the first time one of the commands is invoked. There is also an `autoxload` for ADS applications.
`startapp` str [str]	Takes strings representing a Windows application and a filename and runs the application on the file. It returns a positive integer if it succeeds, 0 if it fails.
`xload` string [obj]	Takes the name of a compiled ADS application and loads it into memory.
`xunload` string [obj]	Takes the name of an ADS application and unloads it from memory.

Memory Management

`alloc` int	Sets the segment size to the specified number of nodes and returns the previous segment size.
`expand` int	Requests (and returns) the specified number of segments from the heap.
`gc`	Forces the garbage collector to run and returns `nil`.
`mem`	Prints current memory usage on the screen and returns `nil`.
`vmon`	Turns on virtual function paging and returns `nil`.

Index

All AutoLISP functions are indexed individually as well as under "function definitions." All programs shown in the text of the book are indexed under "programs." All system variables are indexed individually as well as under "system variables."

!, 40, 181
&, &=, 479
', 176
*, 102
Cancel, 504
error, 524–526
+, 100
-, 101
/
  division, 102–103
  errors, 513
  parameter list usage, 190
  pathname separator, 20
/=, 238
1+, 101
1-, 102
1>, 185, 507
;, 37
;|, 542–543
<, 238
<=, 238
=, 237
>, 238
>=, 239
\, 20, 206, 329
\n, 329
\r, 329
\t, 329
~, 113

A
abbreviating cars and cdrs, 48–49
acad system environment variable, 351–354
acad.lsp, 170, 561–563
acad.pgp, 621–623
acad.unt, 373
ads, 414
AHED editor, 618–621
alert, 523
alert box, 523
aliases, 623
alloc, 15
and, 244
AND, 592
  with boole, 116
  with ssget, 480–483
ANGBASE, 316
ANGDIR, 316
angle, 382
angles
  conversion, 374–376
  inputting, 316–320
angtof, 376
angtos, 374
anonymous blocks, 445
anonymous functions, 301–302
APERTURE, 383, 385
append, 289–290, 510
application-handling functions, 413–414
application ID, 451
apply, 297, 301
AREA, 204
arguments, 35, 131, 132, 136, 158, 188
arithmetic functions, 99–121
arrows, 16–18, 21
ascii, 363, 368–371
ASCII, 596
assoc, 294
associative dimensions, 488

association lists, 293–297, 422
atan, 108
atof, 369
atoi, 368
atom, 227
atoms, 26
atoms-family, 391–396
AUNITS, 374, 376
AUPREC, 374
AutoCAD commands
  creating, 202, 208
  using, 203
AutoCAD library path, 351–352
autoload, 562
autoxload, 563
auxiliary storage devices, 585

B
backbone, 31
binary, 588–591
bit, 587
bit value, 322, 590
BLIPMODE, 198
blocks, 426, 477
  accessing entities within, 456–458
  anonymous, unnamed, 445, 477
  associative dimensions, 489
  attributes, 426
  creating with entmake, 444, 445
  exploding, 461, 522
body, 155
boole, 114
Boolean algebra, 592, 593–594
bound symbols, 16, 229
boundp, 229, 556
breakpoints, 556
byte, 587

C
c:, 209
caddr, 50
cadr, 48–50
car, 21–26, 31, 44–48
CDATE, 415
cddr, 48–50
cdr, 21–31, 44–48
central processing unit (CPU), 583
chr, 369, 491
CLAYER, 498
close, 336
CMDECHO, 199, 206
command, 203
commands, 602
  creating AutoCAD, 208
  modifying AutoCAD, 343–345
comments, 37, 542–543
compiling programs, 608–610
complex entities, 424–427
cond, 247–250
conditionals, 240–250
cons, 41–43
conses, 21–25
  printed representation, 62–63
contract, 36, 510
control characters, 328–329
conversions
  angles, 374–376
  numbers, 109–110, 371–373
  points, 377
  strings, 368–370
  symbols, 370
cos, 108
CVPORT, 398
cvunit, 373

D
debugging, 526–537, 610–611
defun, 154–169, 517, 557–558
  examples, 164–169
dictionaries, 494–496
dictnext, 495
dictsearch, 495
dimensions, 488
display and device functions, 399–412
distance, 382
distof, 372
dotted pairs, 62
draw.lsp, 91, 146
dtr, 107, 164
dynamic scoping, 195

E
editing programs, 605, 608
editor
  AHED, 608, 618–621
  PFE, 608
efc.lsp, 144
elements, 30
endblk, 444
entdel, 433
entget, 435, 453
entities
  complex, 424–427
  data, 423
  group codes, 423, 697–698
  name, 422
  printed representation, 422–423
  within blocks, 455–457
entlast, 428

entmake, 442–446, 486
entmod, 440, 458
entnext, 428
entsel, 429–431, 523
entupd, 441
eq, 231–234
equal, 234–237
errors, 132–134, 504–537
  "AutoCAD rejected function", 514
  "AutoLISP stack overflow", 254
  "bad argument type", 35, 132, 161, 174, 510–511
  "bad configuration variable name", 547
  "bad formal argument list", 134, 509
  "bad function", 134, 175, 507–508
  "Can't reenter AutoLISP", 514
  "divide by zero", 103, 247, 513
  "exceeded maximum string length", 20, 52
  "extra right paren", 52, 507
  "file not open", 336
  "Function cancelled", 548
  "incorrect number of arguments", 35, 132, 161, 512
  "insufficient node space", 598, 600
  "insufficient string space", 598, 602
  "invalid dotted pair", 13, 63, 205, 515
  "LOAD failed", 52, 505
  "malformed list", 52, 506
  "malformed string", 52, 507
  "null function", 134, 509
  "quit / exit abort", 524
  "too few arguments", 35, 132, 512, 520
  "too many arguments", 35, 132, 512
  "Unknown command", 522
  Cancel, 504
  gotchas, 515–523
  sporadic, 535
error handling functions, 523–526, 533–534
eval, 127–132
  examples, 134–144
evaluation, 127–144, 163, 178
Exclusive OR, 592
exit, 524
exp, 107
expand, 601
expt, 105
extended entity data, 450–455, 493
  filtering, 478
extrusion, 424

F

file descriptor, 334
file I/O, 334–350
files, 596
FILLETRAD, 198
filters, 475–478
findfile, 353
fix, 110
float, 109, 513
floating point imprecision, 13, 235
floating point numbers, 12
flowcharts, 606–607
foreach, 258–265
form, 127, 176
free reference, 195
functions, 34–37, 126, 602, 611–615
  arguments, 35
  contract, 36, 38–39
  errors, 510–515
  printed representation, 61, 159
function definitions
  *, 102
  error, 524–526
  +, 100
  -, 101
  /, 102
  /=, 238
  <, 238
  <=, 238
  =, 237
  >, 238
  >=, 239
  ~, 113
  1-, 102
  1+, 101
  abs, 107
  ads, 414
  alert, 523
  alloc, 601
  and, 244
  angle, 382
  angtof, 376
  angtos, 374
  append, 289
  apply, 297
  ascii, 369
  assoc, 294
  atan, 108
  atof, 369
  atoi, 368
  atom, 227
  atoms-family, 391
  autoload, 562
  boole, 114
  boundp, 229
  caddr, 50
  cadr, 48
  car, 44
  cddr, 48
  cdr, 44

function definitions (*cont.*)
chr, 369
close, 336
command, 203
cond, 248
cons, 41
cos, 108
cvunit, 373
defun, 154–156
dictnext, 495
dictsearch, 495
distance, 382
distof, 372
entdel, 433
entget, 435
entlast, 428
entmake, 442
entmod, 440
entnext, 428
entsel, 429
entupd, 441
eq, 233
equal, 234
eval, 127–132
exit, 524
exp, 107
expand, 601
expt, 105
findfile, 353
fix, 110
float, 109
foreach, 258
gc, 599
gcd, 104
getangle, 316
getcfg, 547
getcorner, 314
getdist, 312
getenv, 353
getfiled, 346–350
getint, 310
getkword, 321, 323
getorient, 317
getpoint, 313
getreal, 312
getstring, 314
getvar, 197
graphscr, 397
grclear, 400
grdraw, 400
grread, 406
grtext, 404
grvecs, 401
handent, 449
help, 566
if, 240
initget, 321
inters, 378
itoa, 371
lambda, 301
last, 284
length, 284
list, 43
listp, 227
load, 51, 341
log, 107
logand, 112
logior, 113
lsh, 111
mapcar, 299
max, 106
mem, 600
member, 292
menucmd, 411
min, 106
minusp, 230
namedobjdict, 494
nentsel, 432
nentselp, 432
not, 246
nth, 285
null, 228
numberp, 226
open, 334
or, 245
osnap, 379
polar, 381
prin1, 332
princ, 332
print, 50, 332
progn, 243
prompt, 331
quit, 524
quote, 171
read, 369
read-char, 330
read-line, 327
redraw, 397, 434
regapp, 451
rem, 104
repeat, 251
reverse, 286
rtos, 371
set, 386
setcfg, 546
setfunhelp, 565
setq, 39, 180
setvar, 197
sin, 108
snvalid, 492
sqrt, 106
ssadd, 465
ssdel, 467
ssget, 471
sslength, 466
ssmemb, 468
ssname, 468
startapp, 414
strcase, 362
strcat, 360
strlen, 362
subst, 291
substr, 363
tablet, 412
tblnext, 484
tblobjname, 492
tblsearch, 486
terpri, 331
textbox, 446
textscr, 397
trace, 533
trans, 377

type, 230
untrace, 534
ver, 551
vmon, 600
vports, 398
wcmatch, 365
while, 252
write-char, 331
write-line, 328
xdroom, 454
xdsize, 454
xload, 413
xunload, 413
zerop, 229
fuzz factor, 235–236

G
garbage collection, 599
gc, 599
gcd, 104
getangle, 316–320
getcfg, 547
getcorner, 314
getdist, 312, 488
getenv, 353
getfiled, 346–350
getint, 310
getkword, 321, 324–325
getorient, 317–320
getpoint, 313
getreal, 312
getstring, 314
getvar, 197
global variables, 186–187, 191–192, 201, 543–548
graphics functions, 396–399
graphscr, 397
grclear, 400
grdraw, 400
grounded lists, 33–34
group codes, 423
grread, 406–409
grtext, 404–406
grvecs, 401–403

H
handent, 449
handles, 448–450
heap, 595
help, 566
help facility, 563–573
HIGHLIGHT, 198, 547
hot spots, 571
hypertext links, 571

I
I/O functions, 310–334
if, 240–242
Inclusive OR, 592
indentation conventions, 244, 540
initget, 321–327, 430–433
integers, 12
interpreting programs, 609–610
inters, 378
IOR, 592
iteration, 250–265, 615–616
itoa, 371

K
keywords, 321
  localized, 325–326
kilobyte (kb), 591

L
lambda, 301
last, 284
layers, 424, 438, 485
  making, 493
length, 284
library path, 351–352
LIMCHECK, 322
LIMMAX, 197
list, 43
listp, 227
lists, 26–34
  elements, 30
  functions, 280–303
  grounded, 33–34
  length, 31
  members, 30
  printed representation, 64–74, 83–86
  reduction rules, 83–86
load, 51, 341–343, 347, 561
  errors, 505–507
local variables, 186–191, 202, 258, 543
localized keywords, 325–326
log, 107
logand, 112
logical operators, 478
logior, 113
loops, 250–265, 615–616
lsh, 111
LUNITS, 371, 372
LUPREC, 371

M
max, 106
mapcar, 299, 301
measurement functions, 378–386
megabyte (mb), 585, 591
mem, 600
member, 292
members, 30
memory, 583–585
  nodes and segments, 598–599, 601
memory management, 598–602
menucmd, 411–412
min, 106

minusp, 230
multiprogramming, 584

N
namedobjdict, 494
nentsel, 432, 456
nentselp, 432, 457
nil, 16, 18–19, 27–30, 33–34
node, 598
not, 246–247
NOT, 593
  with boole, 119
  with ssget, 480
nth, 285
null, 228, 555
numberp, 226
numbers, 12–14
  arithmetic functions, 99–120
  binary and decimal, 588–591
  conversions, 109–110
  printed representation, 60

O
object, object type, 10
open, 334, 352
operand, 86, 130, 141, 162, 201
operator, 86, 130, 133, 141, 201
or, 245
OR, 592
  with ssget, 482
OSMODE, 113
osnap, 379–381

P
page table, 600
paging, 597–600
parameters, 157–158, 163, 187–189, 201, 517
pathnames, 20, 51
pause, 205
PERIMETER, 204
pi, 14
PICKBOX, 383
PICKFIRST, 474
points, 34
  comparing, 235–236
  inputting, 313–314
polar, 381
polyline
  creating with entmake, 443
  entity data, 424–427
  header record, 427, 480
  vertex, 425
predicates, 226–239
prefix notation, 86–90
prin1, 332–333
princ, 332–333
print, 50, 332–333
printed representation
  conses, 62–63
  entity names, 423
  file descriptors, 334
  functions, 61, 159
  lists, 64–74
  numbers, 60
  selection sets, 464, 466
  strings, 60
  symbols, 61
progn, 243
programs, 602
  bounds, 262
  box, 207
  boxit, 447
  boxit2, 550
  bp, 533
  c:att-ed, 469
  c:box2, 209
  c:close-files, 394
  c:dimsize, 489
  c:draw, 91, 146
  c:ed-title, 361
  c:efc, 145
  c:find-circles, 436
  c:get-parts, 355
  c:line, 344
  c:long-dist, 255, 541
  c:midln, 383
  c:mov, 211
  c:new-style, 487
  c:pass, 410
  c:printit, 339
  c:sclblk, 477
  c:sl, 438
  c:sparse, 364
  c:tbox, 447
  c:xdadd, 454
  c:zapall, 475
  circ, 260
  counter, 329
  dtr, 164
  factorial, 270
  find-last, 268
  getename, 431
  getpnt, 327
  len, 265
  lin, 298
  log1, 482
  log2, 483
  math, 298
  midpoint, 303, 385
  round, 242
  rsetv, 388
  setv, 388
  swap, 166
  tan, 168
prompt, 331
prompt string, 310

Q
queue, 595
quit, 524
quote, 171–179, 512, 517

R
radians, 164
random-access memory (RAM), 584–585
read, 341, 369
read-char, 330
read-eval-print loop, 182–184
read-line, 327
read-only memory (ROM), 584
real numbers, 12–13
recursion, 130, 265–271, 617–618
redraw, 397, 434
reduction rules, 83–86
regapp, 451
relational tests, 478
rem, 104
repeat, 251–252
reverse, 286
round, 110
rtd, 107, 218
rtos, 371

S
s::startup, 343, 559
SAVETIME, 198
scientific notation, 13
scoping of variables, 197
SCREENBOXES, 403
segment, 599
selection sets, 464–483
  filters, 475–478
  logical operators, 480
  modes, 472
  printed representation, 464, 466
  relational tests, 478
seqend, 427
set, 386, 395, 577
setcfg, 546
setfunhelp, 565
setq, 39, 128, 179–180, 519, 544, 553
setvar, 197
sin, 108
snvalid, 492
special operators, 154
sqrt, 106
ssadd, 465
ssdel, 467
ssget, 470–478
  modes, 472
sslength, 466
ssmemb, 468
ssname, 468
stack, 595
startapp, 414
strcase, 362
strcat, 360
strings, 19–20
  conversion, 368–370
  functions, 360–365
  inputting, 314–315
  printed representation, 60
strlen, 362
STYLE, 488
styles, 487
subentities, 424–427
subrs, 37. *See also* functions
subst, 291, 296, 439
substr, 363
symbol tables, 484–494
symbols, 14–18, 199–202
  bound and unbound, 16
  conversions, 371
  functions, 179–180, 386
  global. *See* global variables
  local. *See* local variables
  name, 14–15
  printed representation, 61
  value, 16–18, 127–128, 163
system environment variables, 350–351
system variables, 197, 389, 547
  ANGBASE, 316–317
  ANGDIR, 316–317
  APERTURE, 383
  AREA, 204
  AUNITS, 374, 376
  AUPREC, 374
  BLIPMODE, 198
  CDATE, 415
  CLAYER, 498
  CMDECHO, 198, 206
  CVPORT, 398
  FILLETRAD, 198
  HIGHLIGHT, 198
  LIMCHECK, 322
  LIMMAX, 197
  LUNITS, 371, 372
  LUPREC, 371
  OSMODE, 112
  PERIMETER, 204
  PICKBOX, 383
  PICKFIRST, 474
  SAVETIME, 198
  SCREENBOXES, 403
  STYLE, 488
  USERR1–5, 546
  VIEWSIZE, 276

T
t, 18–19, 521
tables, 484–494
tablet, 412
tan, 108, 168–169
tblnext, 484
tblobjname, 492
tblsearch, 486
terpri, 331
text styles, 487
textbox, 446
textpage, 397
textscr, 397

timesharing, 584
trace, 533
trans, 377
transformation matrix
  grvecs, 402
  nentsel, 457–458
  nentselp, 457
trigonometric functions, 107–108, 168–169
type, 230

U
unbound symbols, 16
unnamed blocks, 445
untrace, 534
USERR1–5, 546

V
variables, 17, 200, 551
  global. *See* global variables
  local. *See* local variables
  scoping, 197
vector, 401
ver, 551
vertex, 425
viewports, 398, 486
VIEWSIZE, 276
virtual function paging, 600
virtual memory, 597–598, 600
vmon, 600
vports, 398

W
wcmatch, 365
while, 252–258
wildcards, 365–367
  with ssget, 476
word, 589–591
write-char, 331
write-line, 328

X
xdata, 450–455, 494
  filtering, 478
xdroom, 454
xdsize, 454
xload, 413
XOR, 592
  with boole, 118, 119–120
  with ssget, 481
XOR ink, 400, 401
xunload, 413

Z
zerop, 229

About the Author

Roy Harkow is the president of Roy Harkow Associates, an organization that specializes in AutoLISP and Common Lisp training. He has been teaching Lisp since 1983, and since 1988 has been presenting AutoLISP workshops at Authorized Training Center locations and companies throughout the United States and internationally. He is a frequent contributor to *CADENCE* magazine and *AutoCAD Tech Journal*.

Roy has a B.A. in English from SUNY Albany and an M.S. in Computer and Information Science from Ohio State University. He has programmed computers at IBM, Unisys, and Digital Equipment Corporation and has taught computer programming at DEC, Symbolics, and Gold Hill Computers.

For information regarding workshops and AutoLISP consulting, please call Roy Harkow Associates at 617-237-8870 from 10-6 EST daily. Roy welcomes questions and comments about *Essential AutoLISP,* as well as corrections, interesting new labs, and suggestions for future releases and revisions. He can be reached via Internet at `roy@yem.pn.com`, or you can write to him at Roy Harkow Associates, 40 MacArthur Road, Wellesley, MA 02181. Please include your return address, fax, and phone number.